MASTER EN SCIENCES INFORMATIQUES

Rétro-Ingénierie des Applications de Gestion

Bodart, Stéphane; Lebrun, Laurent

Award date:
1995

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Rétro-Ingénierie des Applications de Gestion

Stéphane BODART
Laurent LEBRUN

Promoteurs :
Jean-Luc HAINAUT et Naji HABRA

Mémoire réalisé en vue de l'obtention du grade de Licencié et Maître en Informatique

Année académique 1994 - 1995

Résumé

Le but de ce mémoire est de concevoir un méta-schéma permettant la modélisation de tout type d'applications et ce, à tout niveau d'abstraction. Ce méta-schéma doit être capable de représenter aussi bien la partie relative à la définition des données que celle relative à la définition des traitements d'une application.

Une analyse de différents modèles nous a permis de dégager les concepts nécessaires à sa création. La partie du méta-schéma concernant les données a déjà été conçue au sein de l'atelier DB-Main. Cependant, nous indiquerons certaines modifications qui devraient y être apportées. Nous complèterons ce schéma en y ajoutant des structures capables de représenter la partie traitements d'un modèle.

Ce mémoire inclut encore la proposition d'une interface concernant la saisie et l'affichage des informations ayant rapport aux traitements.

Abstract

The aim of this master thesis is to conceive a meta-schema allowing the modelisation of any kind of applications, at different abstraction levels. This meta-schema must be able to represent the data definition part as well as the process definition part of an application.

The necessary concepts usefull for its creation have been found after an analysis of different models. The data definition part of the meta-schema has already been designed within the DB-Main CASE tool. However, we'll indicate some modifications that should be implemented in this meta-schema. We'll complete this schema by adding structures which are able to represent the process part of a model.

This thesis also includes the proposition of an interface allowing to get and to display informations about processes.

Remerciements

Nous tenons à exprimer nos profonds remerciements à nos promoteurs, Messieurs Jean-Luc Hainaut et Naji Habra, pour nous avoir guidés tout au long de ce travail et pour les conseils qu'ils nous ont fournis.

Nous remercions également l'équipe de chercheurs du projet DB-Main pour les éclaircissements qu'ils nous ont apportés.

Nous ne pouvons oublier de remercier les professeurs David Wastell et Bryan Warboys qui nous ont accueillis durant un semestre à l'Université de Manchester. Le travail réalisé avec l'aide de Martyn Spink, Peter Kawalek et Ian Robertson a été très enrichissant et restera une expérience positive.

Nous tenons également à exprimer notre gratitude à l'ensemble des professeurs et assistants qui nous ont fourni des renseignements indispensables à la réalisation de notre mémoire.

Table des matières

Table des figures

Table des exemples

1. Introduction

Le but de ce mémoire est de concevoir un méta-schéma permettant la modélisation de tout type d'applications et ce, à tout niveau d'abstraction. Ce méta-schéma doit être capable de représenter aussi bien la partie relative à la définition des données que celle relative à la définition des traitements.

La première étape de notre recherche a consisté en une sélection de différents modèles couvrant les trois niveaux d'abstraction : le niveau conceptuel, le niveau logique et le niveau physique. Le second chapitre a donc été consacré à une présentation et une analyse de chacun de ces modèles afin d'en extraire les concepts fondamentaux que nous aurons à représenter.

Le chapitre suivant est consacré à l'élaboration proprement dite du méta-schéma. Pour ce faire, nous commençons par faire une synthèse des concepts identifiés au chapitre 2. Ceux-ci se regroupent en deux catégories : les données et les traitements. En ce qui concerne la partie données, un méta-schéma a déjà été élaboré par l'équipe de recherche du projet DB-

Main. Une section est donc consacrée à sa présentation. Malheureusement certains concepts relatifs à la partie données ne peuvent y être représentés. C'est pourquoi une section est consacrée à une critique de ce méta-schéma. Nous proposerons ensuite quelques modifications qui nous semblent utiles pour remédier à ces problèmes, ainsi qu'une extension permettant la modélisation des concepts relatifs à la partie traitements.

Le chapitre 4 est, quant à lui, consacré à la validation de ce méta-schéma. Nous allons donc reprendre l'ensemble des modèles analysés au chapitre 2 afin de montrer comment ils peuvent s'y insérer.

Le chapitre 5 traitera un programme complet. Nous avons choisi d'analyser un programme Cobol étant donné que de nombreuses applications actuelles ont été réalisées dans ce langage.

Ce méta-schéma devant être implémenté au sein de l'outil CASE DB-Main, le dernier chapitre sera consacré à la présentation d'une interface. Afin de rester cohérent avec la partie déjà implémentée, nous commencerons par présenter les boîtes de dialogue existantes avant d'en proposer de nouvelles destinées à supporter la partie traitements des applications.
La dernière section de ce chapitre consistera à développer une représentation textuelle et graphique de l'ensemble des informations relatives aux traitements.

2. Analyse du problème

2.1 Introduction

Afin d'avoir une idée plus précise des types d'objets que nous allons devoir représenter, nous allons faire une petite étude de différents modèles. La section suivante de ce chapitre est donc consacrée à l'élaboration d'une grille d'analyse. Il est bien sûr évident que nous ne couvrirons pas l'ensemble des modèles existants, mais les modèles sélectionnés nous semblent représenter une assez large variété de types de modèles.

Dans la dernière section, nous en ferons une présentation avant d'en dégager les différents concepts présents.

2.2 *Grille d'analyse*

Comme nous l'avons déjà dit, cette section va nous permettre d'établir une grille d'analyse de différents modèles. Ceux-ci ont été sélectionnés et jugés représentatifs des diverses familles de modèles. Nous avons tenté d'en retenir à différents niveaux d'abstraction allant du niveau conceptuel au niveau physique.

Cette sélection nous permettra de déduire les concepts fondamentaux que nous y trouvons.

Niveau conceptuel	Modèle des traitements	Dataflow Diagram	Types Abstraits	Télos	Coad & Yourdon
Niveau logique	V.D.M.	OBLOG			
Niveau physique	C	C++	Pascal	PML	

Nous allons commencer notre analyse par le niveau conceptuel et, par la suite, descendre jusqu'au niveau physique.

2.3 *Analyse des différents modèles*

2.3.1 La méthode Merise

2.3.1.1 Présentation[1]

Merise est une méthode de conception d'un système d'information. Elle a été développée dans le but de diminuer l'ampleur des difficultés liées à la conduite d'un projet informatique.
Pour ce faire, Merise lie la mise en place du système informatisé avec la refonte de l'organisation en travaillant sur une vue globale de l'entreprise.
Merise est également caractérisée par une approche par niveaux. En effet, on y distingue trois niveaux de réflexion :

- niveau conceptuel :

  "Il correspond à la définition des finalités de l'entreprise en explicitant sa raison d'être".
- niveau organisationnel :

  "Son rôle est de définir l'organisation qu'il est souhaitable de mettre en place dans l'entreprise pour atteindre les objectifs fixés".
- niveau technique :

  A ce niveau, on intègre les moyens techniques nécessaires au projet.

"Selon qu'ils s'appliquent aux données ou aux traitements, ces niveaux seront modélisés sous des qualificatifs différents".

Niveaux	Données	Traitements
Conceptuel	Modèle conceptuel des données	Modèle conceptuel des traitements
Organisationnel	Modèle logique des données	Modèle organisationnel des traitements
Technique	Modèle physique des données	Modèle opérationnel des traitements

[1] La présentation de ce modèle est extraite de [COL86].

"La séparation des données et des traitements jointe à la définition des niveaux permet d'aborder successivement les problèmes et de se situer à tout moment dans l'avancement des travaux".

Une première étape précède ces différentes phases ; il s'agit de l'étude de l'existant. De plus, des phases de validation se trouvent intercalées.

"Le parcours des différents niveaux peut être chronologiquement le suivant :

1) Etude de l'existant

2) (En parallèle, par deux équipes différentes, si cela est possible)

Modèle conceptuel des données	Modèle conceptuel des traitements
	Modèle organisationnel des traitements

3) Validation

4) Modèle logique des données

5) (Ensemble)

Modèle physique des données

Modèle opérationnel des traitements"

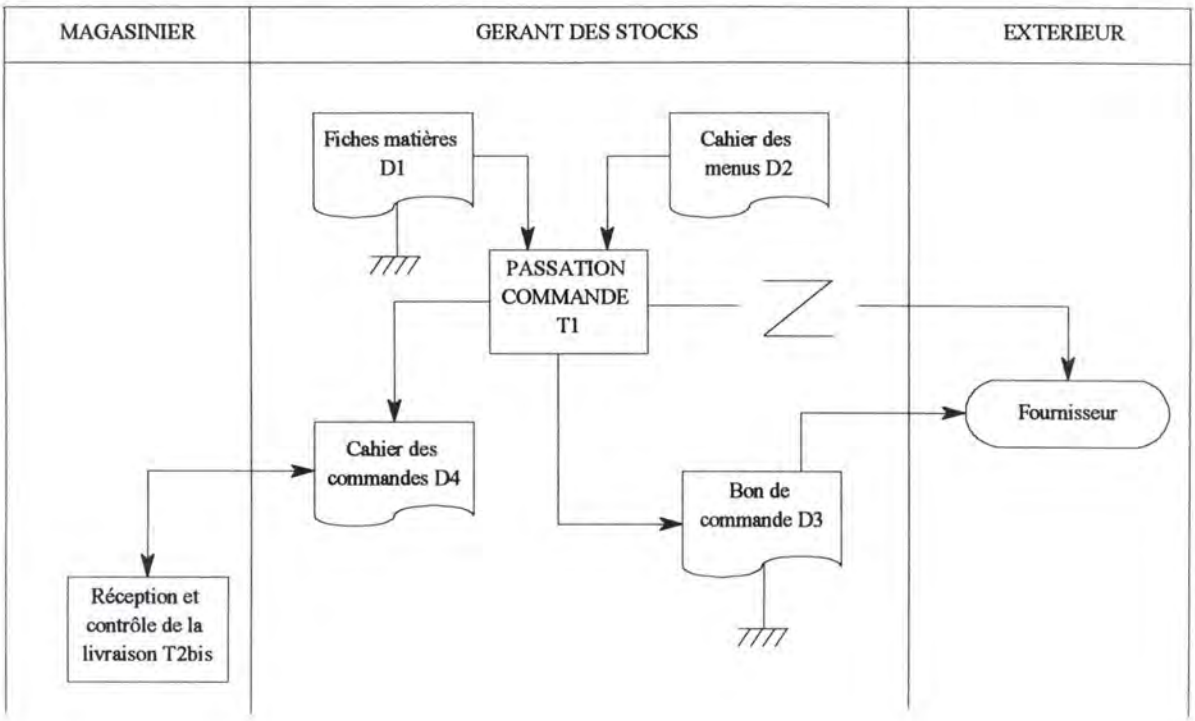
Nous n'allons pas analyser en détail chacune de ces étapes car cela sortirait du cadre de ce mémoire. Nous allons juste étudier par quelques exemples les différents formalismes utilisés. Chaque étape n'a pas son formalisme propre. En effet, certaines représentations sont communes à plusieurs étapes.

Exemple :

- ***Etude de l'existant***

L'étude de l'existant se fait principalement au moyen d'interviews. Au fil de l'interview, on construit un diagramme tâches-documents. Ce diagramme nous montrera l'enchaînement des différentes tâches en fonction des documents produits.

Voici un exemple de ce diagramme, représentant l'extrait d'une interview d'un responsable de la gestion des stocks (voir Exemple 2-1).



Exemple 2-1: Merise - Etude de l'existant

Explication du formalisme :

Tâche

Document

Poste de travail souvent exclu du champ de l'étude

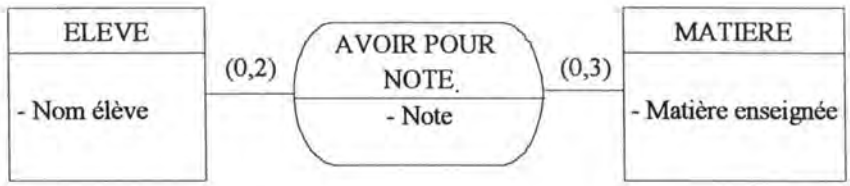
Classement du document

Transmission par téléphone, télex, ...

A ce diagramme, on attache un autre document qui décrit de manière plus précise le contenu des tâches et des documents.

- ***Modèle conceptuel des données***

Au niveau du modèle conceptuel des données, un autre formalisme est utilisé (voir Exemple 2-2).

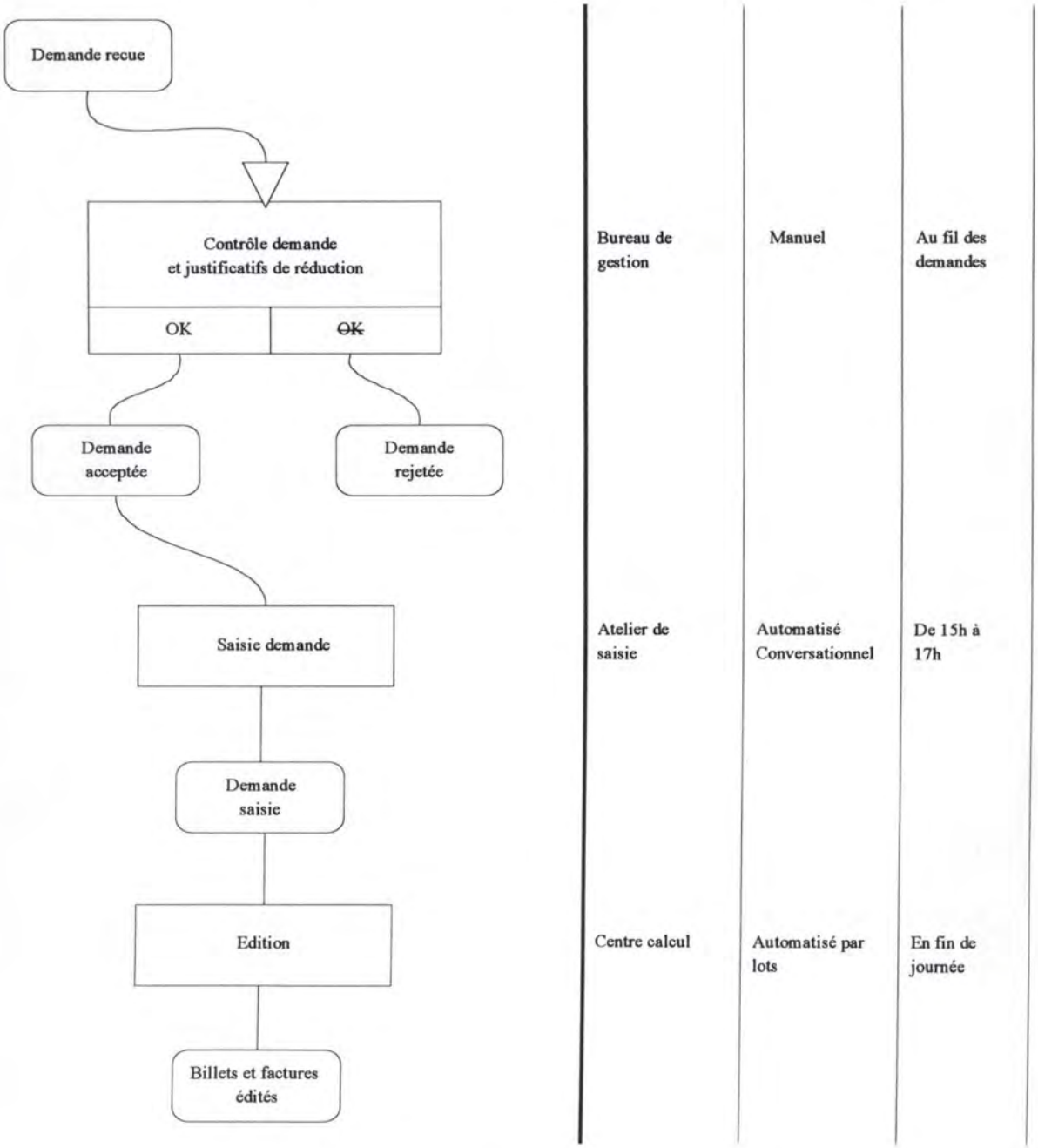


Exemple 2-2 : Merise - Modèle conceptuel des données

Nous avons deux objets : ELEVE et MATIERE. Ces objets participent à la relation AVOIR POUR NOTE. Les cardinalités indiquent qu'un élève participe au minimum 0 fois à la relation et un maximum de 2 fois, et qu'une matière y participe minimum 0 fois et maximum 3 fois.

- ***Modèle organisationnel des traitements***

La représentation du niveau conceptuel des traitements est fort similaire à celle utilisée au niveau organisationnel des traitements. La différence principale réside dans le fait que dans ce dernier, nous précisons, en plus, pour chaque opération le poste de travail, la nature du traitement et sa chronologie de déclenchement.
Nous n'analyserons que le cas le plus global (voir Exemple 2-3).



Exemple 2-3 : Merise - Modèle organisationnel des traitements

Ces trois représentations nous paraissent suffisantes pour tenter de déterminer les différents concepts que nous devons pouvoir représenter.

2.3.1.2 Recherche des concepts

- ***Etude de l'existant***

  Les concepts présents à cette étape sont principalement ceux de tâche et de document. Il nous faudra donc un moyen de les représenter, ainsi qu'un moyen d'indiquer les flux de ces documents. Nous devrons également pouvoir associer un acteur à ces deux objets.

- ***Modèle conceptuel des données***

  Nous ne trouvons dans cette représentation que des concepts classiques aux schémas entité-association : entité, relation, cardinalité, ... Il nous faut donc pouvoir représenter ces objets.

- ***Modèle organisationnel des traitements***

  Les concepts sont relativement proches de ceux présents au niveau de l'étude de l'existant, du moins du point de vue de la représentation. Nous trouvons le concept de phase (correspondant à une action), d'évènement et de résultat.

  Nous devons pouvoir représenter les liens entre tous ces éléments, ainsi que des informations supplémentaires relatives aux phases (poste de travail, nature du traitement et sa chronologie de déclenchement).

2.3.2 Dataflow Diagram

2.3.2.1 Présentation[2]

Il s'agit d'un langage de spécification qui a été développé à la fin des années 1970. C'est un langage à caractère graphique.

[2] La présentation et l'exemple repris dans cette section sont tirées de [MDL94].

Nous y trouvons les éléments suivants :

pour représenter une fonction ;

pour représenter une donnée qui circule ;

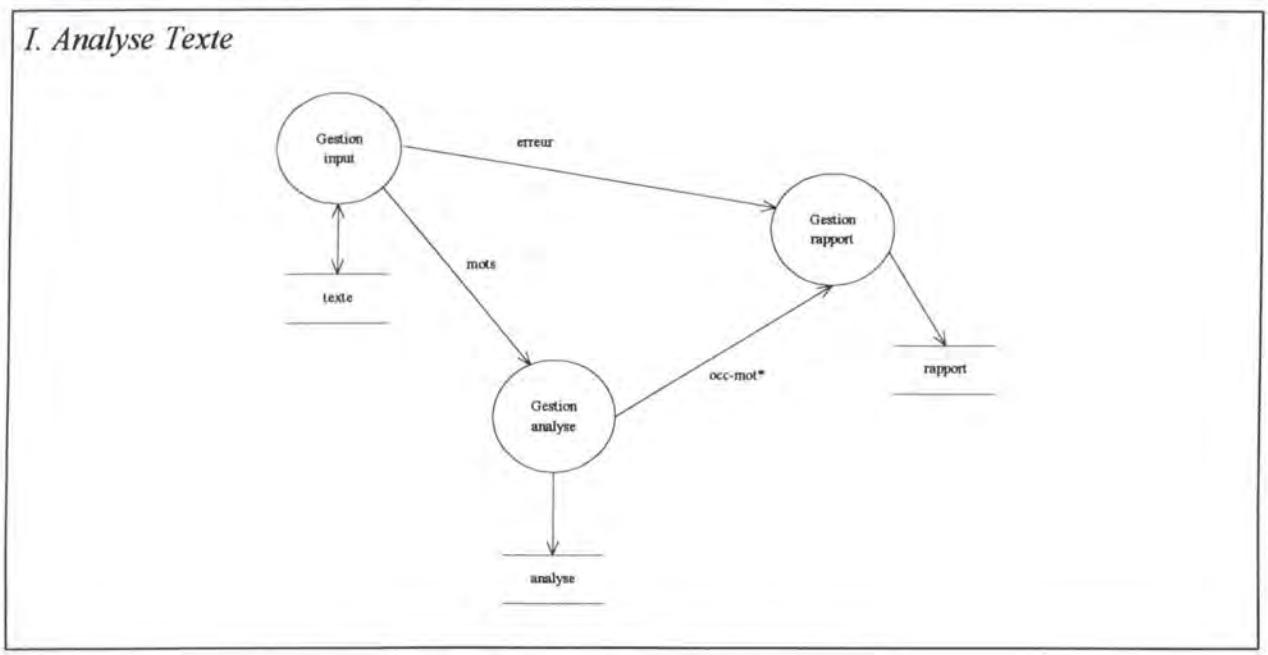
pour représenter une information persistante.

Ce langage permet de travailler par raffinements successifs. Lorsque l'on a atteint un niveau de spécification jugé suffisant, il suffit de représenter les différentes fonctions restantes. Cela peut se faire de différentes manières : algorithme, pseudo-code, ...

Dans l'exemple ci-dessous, les fonctions de plus bas niveau sont spécifiées au moyen de pré et postconditions en utilisant un pseudo-code (voir Exemple 2-6).

Exemple : Le but de cet exemple est de calculer le nombre d'occurrences de chaque mot dans un texte et de rédiger un rapport contenant les résultats.

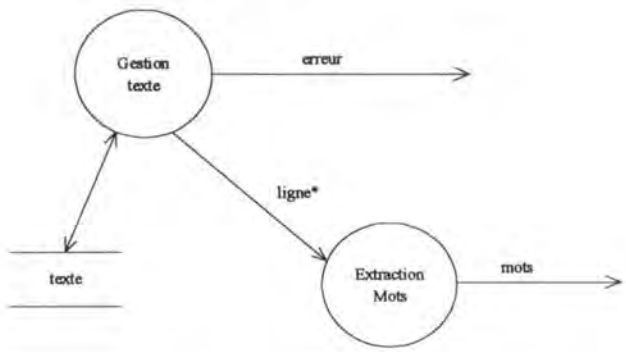
Une vue globale du problème peut être schématisée comme suit :



Exemple 2-4 : DFD - Définition d'une action

Chacune de ces trois fonctions peut être raffinée. Cependant, seul le cas de la fonction "Gestion input" (voir Exemple 2-5) va être étudié par la suite.

I.1. Gestion Input



Exemple 2-5 : DFD - Raffinement d'une action

Les deux fonctions restantes "Gestion texte" et "Extraction Mots" peuvent encore être raffinées, mais nous avons choisi d'en rester à ce niveau.

Nous allons donc montrer comment la fonction "Gestion texte" peut être représentée à ce stade :

```
(out, texte') = Gestion-texte(texte)

    LIGNE = SEQ[CHAR]
    texte, texte', l, t, t', t'' : SEQ[LIGNE]

    PRE
    POST
        out = 'Fichier vide' <= Eof(texte)
        (out*, texte*) = Extraction(texte) <= not eof(texte)

        (t', l*) = Extraction(t)
            <=> IF Eof(t)
```

```
                THEN l* = [] and t'=t
                ELSE (t'', l) = Read(t)
                        l* = Append(l, Extraction(t''))
```

Exemple 2-6 : DFD - Représentation textuelle d'une action

2.3.2.2 Recherche des concepts

Les concepts présents dans ce langage sont assez classiques. Il s'agit des concepts de fonction et de donnée (persistante ou non). Il est important de pouvoir représenter les relations entre ces fonctions et ces données car c'est ce qui caractérise ce langage.
Il faudra également pouvoir stocker la définition des fonctions de plus bas niveau qui peut se faire, comme nous l'avons déjà dit, selon différents formalismes.

En ce qui concerne les données, nous devons être à même de représenter des variables. Ces variables peuvent être de type standard (tels des entiers, des caractères, ...) ou de type non standard (tels des listes, des séquences, ...) Nous devrons également pouvoir représenter des données persistantes.

2.3.3 Types abstraits algébriques

2.3.3.1 Présentation[3]

Les types abstraits algébriques ont été créés par Liskov et Guttag.

Ils constituent une méthode de conception orientée objet (basée sur les structures de données manipulées plutôt que sur les fonctions assurées par le système).

Un type abstrait algébrique permet d'éviter le choix de la structure de données et de décrire les opérations sans faire référence à leur effet sur la structure de données.

[3] Cette présentation provient de [MDL94].

Avantages des types abstraits algébriques :

- Ils évitent un mauvais choix de la structuration de données ;
- Ils reposent sur des fondements mathématiques qui permettent de vérifier automatiquement la complétude et la consistance.

Désavantage :

- Ils utilisent un formalisme mathématique rendant leur utilisation assez lourde.

Il existe trois types d'opération :

- les constructeurs : construisent de nouvelles occurrences du type ;
- les modificateurs : modifient la valeur d'un objet ;
- les observateurs : observent un objet.

Exemple

PileVide : → Pile-Mots
Empiler : Pile-Mots → Pile-Mots
Dépiler : Pile-Mots → Pile-Mots
Tête : Pile-Mots → Mots
Pvide? : Pile-Mots → Bool

Tête (Empiler (pile, m)) = m
Pvide? (Empiler (pile, m)) = Faux
Pvide? (PileVide ()) = Vrai
Dépiler (Empiler (pile, m)) = pile
Tête (PileVide ()) = erreur
Pré : Tête (p) : ¬ Pvide? (p)
Dépiler (PileVide ()) = PileVide
Pré : Dépiler (p) : ¬ Pvide? (p)

Exemple 2-7 : TAA - Spécification du type PILE

La vie d'un objet se représente par la suite des actions qui ont été effectuées sur lui (voir Exemple 2-8).

Dans l'exemple ci-dessus, nous pouvons simplifier l'expression

p = Dépiler (Empiler (Empiler (PileVide, s), s'))

Cela donne

p = Empiler (PileVide, s)

Exemple 2-8 : TAA - Simplification d'une expression

Il existe deux types de relation de structuration : la généricité et l'héritage.

a) La généricité

Exemple

Spécification de FILE [X]. On définit un type abstrait générique qui est une file dont les éléments peuvent être n'importe quel objet (X) (voir Exemple 2-9).

F-Vide : $\rightarrow$ FILE

Arrivée : FILE $\times$ X $\rightarrow$ FILE

Départ : FILE $\rightarrow$ FILE

A-Servir : FILE $\rightarrow$ X

Exemple 2-9 : TAA - Spécification du type FILE

Lorsque l'on voudra utiliser une file, il suffira d'instancier le type abstrait générique (voir Exemple 2-10) :

FILE-CLIENT = FILE [CLIENT]

FILE-FOURNISSEUR = FILE [FOURNISSEUR]

Exemple 2-10 : TAA - Instanciation du type générique FILE

b) L'héritage

Exemple

```
RECTANGLE is CP  (Point1 : POINT,
                  Point2 : POINT,
                  Point3 : POINT,
                  Point4 : POINT)
{CP représente un produit cartésien. On suppose que le type POINT a été défini précédemment.}

Périmètre (r) = p
        r : RECTANGLE
        p : INTEGER

        Pré :
        Post :p = 2 * (côté1 + côté2)
              côté1 = Ord (Point2 (r)) - Ord (Point1 (r))
              côté2 = Abs (Point3 (r)) - Abs (Point2 (r))

CARRE isa RECTANGLE
Invariant : Ord (Point2 (c)) - Ord (Point1 (c)) = Abs (Point3 (c)) - Abs (Point2 (c))

Périmètre (c) = p
        c : CARRE
        p : INTEGER

        Pré :
        Post :p = 4 * côté
              côté = Ord (Point2 (c)) - Ord (Point1 (c))
```

Exemple 2-11 : TAA - L'héritage

Dans cet exemple (voir Exemple 2-11), le type CARRE hérite de la structure et des traitements de RECTANGLE, mais il lui est également possible de redéfinir certaines méthodes (périmètre dans ce cas).

2.3.3.2 Recherche des concepts

Un objet spécifié par les types abstraits est en fait un objet composite comprenant une partie donnée et une partie méthode. Nous allons représenter cela par deux concepts différents reliés entre eux. Le premier est une représentation de la structure de données tandis que l'autre est la représentation des méthodes de l'objet.

Dans la partie données, nous devons représenter la structure des différents objets. Dans la partie traitements, il s'agit de modéliser les actions pouvant être effectuées sur un objet.

2.3.4 Telos

2.3.4.1 Présentation

Telos n'est pas un langage de programmation, mais un langage destiné à représenter formellement des connaissances à propos d'un système d'information.
Pour ce faire, Telos dispose d'une notation formelle et d'un mécanisme de déduction (qui va travailler par inférence sur la base de connaissances) [MYL90].

Telos dispose de 5 opérations pour construire, consulter ou mettre à jour une base de connaissances. Il s'agit des opérations *Tell*, *Untell*, *Retell*, *Retrieve* et *Ask* [MYL90].

Deux types d'unités sont présents dans une base de connaissances :

- "individuals" : qui représentent des entités ;
- "attributes" : qui représentent des relations binaires entre entités.

Ces deux unités constituent une proposition [MYL90].

Trois modes de structuration des entités sont également disponibles :

- L'aggrégation : ce mécanisme permet de construire des objets composés à partir de leurs objets composants [ELM94].

  Ex.: *pièces, portes, fenêtres "are partof" bâtiment* [MYL95]

- La classification : il s'agit de pouvoir classifier des objets similaires dans des classes [ELM94].

  Chaque entité est une instance d'une ou de plusieurs classes.

  Nous avons donc une certaine hiérarchie des classes.

  - *tokens* : propositions qui n'ont pas d'instances et qui représentent des entités ;
  - *simple classes* : propositions qui n'ont que des *tokens* comme instances ;
  - *metaclasses* : propositions qui n'ont que des classes comme instances ;
  - *metametaclasses* : ... ;
  - ... ;
  - *ω classes* : classes de niveau le plus élevé. Elles peuvent avoir comme instance aussi bien des *tokens* que des *simple classes*, *metaclasses*, ... [MYL95]

  Tout objet hérite des attributs et des propriétés de la classe dont il est l'instance.

  Ex. : *maria "instanceOf" personne* [MYL95]

- La généralisation : cela permet de généraliser plusieurs classes en une classe de niveau d'abstraction plus élevé [ELM94].

  Il peut être représenté par une relation ISA.

  Ex.: *étudiant "isA" personne* [MYL95]

Exemple [MYL90]

```
TELL CLASS Paper IN SimpleClass WITH
        attribute
                author : String;
                referee : String;
                title : String;
                pages : 1..100
END
```

Exemple 2-12 : Telos - Définition d'une classe

Nous définissons une classe *Paper* (voir Exemple 2-12) qui est une instance de la classe *SimpleClass*. Nous lui associons une série de paramètres.

```
TELL TOKEN martian IN Paper WITH
        author
                firstauthor     : Stanley;
                                : LaSalle;
                                : Wong
        title
                : 'The MARTIAN system'
END
```

Exemple 2-13 : Telos - Définition d'un token

Cette opération permet de définir un *token* (voir Exemple 2-13). Nous avons différents attributs, ainsi qu'un identifiant externe "martian". Nous voyons que "martian" est une instance de la classe "Paper" définie précédemment.
"Firstauthor" est l'étiquette du premier attribut, et est une instance de l'attribut "author" de la classe "Paper". Le second attribut n'a, quant à lui, aucune étiquette.

Nous pouvons aussi spécialiser la classe "Paper" (voir Exemple 2-14), en ajoutant certains attributs.

```
TELL CLASS AcceptedPaper IN SimpleClass ISA Paper WITH
        attribute
                pages : 1..15
                session : Integer
END
```

Exemple 2-14 : Telos - Spécialisation d'une classe

Telos permet également de spécifier des contraintes d'intégrité et des règles de déduction (voir Exemple 2-15).

```
TELL CLASS Paper IN SimpleClass WITH
        integrityContraint
                :$      (∀ y/Person)
                        (y ∈ this.author ⇒ ¬y∈ this.referee) $
        deductiveRule
                :$      (∀ x/Paper)(∀ z/Address)
                        (z ∈ x.author.address ⇒ z ∈ x.replyAddress) $
END
```

Exemple 2-15 : Telos - Contrainte d'intégrité et règle de déduction

La contrainte d'intégrité veille à ce qu'un auteur ne puisse pas référencer son propre article, tandis que la règle de déduction stipule que l'adresse d'un auteur doit aussi être une adresse de réponse.

La consultation de la base de connaissances peut se faire au moyen des instructions *Retrieve* et *Ask*.

L'instruction figurant à l'Exemple 2-16 nous indique si "LaSalle" appartient ou non aux auteurs repris dans le *token* "martian" :

```
ASK : LaSalle ∈ martian.author.
```

Exemple 2-16 : Telos - Instruction de consultation

2.3.4.2 Recherche des concepts

Le concept central de ce langage est celui de classe. Chaque classe peut être composée d'un certain nombre d'attributs, ainsi que de contraintes d'intégrité et de règles de déduction. Un mécanisme doit donc être trouvé pour le représenter.

Il nous faudra également trouver un moyen pour représenter les relations qui existent entre différentes classes. Il peut s'agir de relations d'héritage, d'instanciation, de spécialisation, ...

La notion de *token* doit également pouvoir être représentée. Ce qui implique que nous devons pouvoir assigner certaines valeurs aux attributs des classes.

2.3.5 Coad & Yourdon

2.3.5.1 Présentation[4]

Cette présentation, plutôt que d'exposer un langage, va développer une stratégie de modélisation d'un problème définie par Coad et Yourdon.

Coad et Yourdon définissent un objet de la façon suivante : un objet est un ensemble d'informations et une description de sa manipulation.
Ils définissent également une stratégie de recherche des objets d'un problème. Elle consiste en cinq étapes :

- Où regarder : la première étape consiste à examiner le domaine du problème, le monde de l'utilisateur ;
- Que chercher : à partir des spécifications obtenues à l'étape précédente, il faut maintenant trouver les structures nécessaires pour représenter le problème, ensuite voir si d'autres systèmes doivent interagir avec le système que l'on veut modéliser. Il faut alors se demander quel matériel devra être utilisé, quels événements doivent être pris en compte, quels sont les rôles joués par les êtres humains, de quels sites le système considéré doit-il avoir connaissance et à quelles unités organisationnelles appartiennent les êtres humains ;
- Que considérer : il s'agit de déterminer si un objet doit réellement être inclus dans le modèle ;
- Que valider : si un objet inséré dans le modèle se révèle par la suite superflu, il faut le supprimer ;
- Comment nommer : les noms des objets devraient être choisis parmi le vocabulaire standard du domaine considéré et non parmi le vocabulaire de l'informaticien. En effet, il est préférable d'utiliser des noms que l'utilisateur peut comprendre aisément.

[4] Les éléments de cette présentation ont été extraits de [COA??].

Il existe également des mécanismes de structuration qui sont la classification et l'assemblage.

- La classification représente une organisation reflétant la généralisation et la spécialiation ;
- L'assemblage représente l'aggrégation, à savoir, la représentation de différentes parties d'une même structure.

En ce qui concerne les attributs, Coad et Yourdon proposent la stratégie suivante :

- Identification des attributs ;
- Positionnement des attributs en utilisant la classification ;
- Identification des connexions entre les instances d'objets et définition des cardinalités ;
- Révision des objets : en ajoutant des attributs, il peut s'avérer nécessaire de modifier des objets créés précédemment ;
- Spécification des attributs : cette étape consiste en une spécification textuelle des attributs (nom et description).

Il nous reste maintenant à définir les services que vont offrir les objets. Un service, selon Coad et Yourdon, est le traitement à effectuer lors de la réception d'un message.
La stratégie de définition des services comprend quatre étapes :

- Identifier les services de base ;
- Identifier des services supplémentaires ;
- Identifier les connexions de messages : cela consiste à identifier les expéditeurs et les récepteurs de chaque message ;
- Spécifier les services : cette partie développe les spécifications des traitements.

2.3.5.2 Recherche des concepts

Dans ce langage, nous sommes confrontés à des objets encapsulant des définitions de données et de traitements. Notre représentation devra donc contenir tous ces éléments. Nous devons dès lors trouver une représentation pour les données définies et une autre représentation pour

les traitements. Ces deux représentations auront des relations entre elles. En effet, les objets utilisés dans un traitement sont des objets définis dans la partie donnée. Pour représenter l'utilisation d'une donnée par un traitement, nous aurons besoin d'une notion de paramètre d'un traitement.

2.3.6 V.D.M.

2.3.6.1 Présentation

VDM (Vienna Development Method) est une méthode de développement formelle, qui comprend un langage de spécification et un système de preuve [FIE92].
Le langage de spécification se base sur des constructions mathématiques (entiers, ensembles, ...). Il est "orienté-modèle" dans la mesure où les opérations du système sont définies en termes de modèles définis sur des types de données abstraits [FIE92].
Trois composants sont présents dans une spécification :

- la définition des entités ;
- la définition des opérations ;
- la définition des fonctions.

Aussi bien les opérations que les fonctions reçoivent des arguments et fournissent des résultats, mais l'opération a en plus la possibilité de travailler sur l'état du système qui est modélisé [FIE92].

Voici les différentes syntaxes utilisées :

- entité :

Rec-name :: fieldname1 : type1
fieldname2 : type2
...

- fonction :

functionname : P1 x P2 x ... → R

functionname (param1, param2, ...) $\triangleq$ *body*

pre *precondition*

post *postcondition*

- opération :

op (p1 : P1, p2 : P2, ...) r : R

ext rd *rs : RS*

wr *ws : WS*

pre *precondition*

post *postcondition* [FIE92]

Exemple : Gestion d'un club d'aviation [GES89].

Nous distinguons trois entités : Pilots, Planes et Flights, que nous définissons de la façon suivante :

Pilots ::	name : *Names;*
	address : *Addresses;*
	license : *License;*
	account : *Accounts;*
	involved : {Yes, No}.
Planes ::	regnum : *RegNums;*
	type : *PlTypes;*
	year : *Years;*
	triprec : *TriRec;*
	hirrate : *HirRates.*
Flights ::	pilot : *Names;*

plane : *RegNums;*

dt : *Dates;*

tt : *Times;*

dl : *Dates;*

tl : *Times;*

invoice : *Names.*

Exemple 2-17 : VDM - Définition d'entités

Dans cet exemple, *Pilots, Planes et Flights* sont des types composés.

En plus de cela, nous avons un système qui est composé de pilotes, d'avions et de vols (voir Exemple 2-18).

Systems *::* pilots : set of Pilots;

planes : set of Planes;

flights : set of Flights;

where inv-systems = ($\forall$p1, p2 $\in$ Pilots . p1 $\neq$ p2 $\Rightarrow$ name(p1) $\neq$ name(p2)) $\wedge$

$\forall$p1, p2 $\in$ Planes . p1 $\neq$ p2 $\Rightarrow$ regnum(p1) $\neq$ regnum(p2))

Exemple 2-18 : VDM - Définition du système

L'invariant indique simplement le fait qu'un pilote est identifié par son nom, et qu'un avion est identifié par son numéro d'enregistrement.

La définition ci-dessus (voir Exemple 2-18) n'est, bien sûr, pas complète. En effet, il faut encore définir les types *Names, Addresses, Licenses,* ... Cependant, cela n'est pas d'un grand intérêt dans le cadre qui nous intéresse.

Il nous reste maintenant à définir certaines opérations. Supposons que l'on désire enregistrer un nouveau pilote dans le système, ainsi que donner pour chaque pilote la liste des vols qu'il a effectués entre deux dates. Nous pouvons le réaliser au moyen des deux opérations suivantes :

- AddPilot
- ListFlightsPilots.

AddPilot (n : Names, a : Addresses, l : Licenses, q : Qualifs)

ext *wr club : Systems*

pre $\forall p \in pilos(club) . name(p) \neq n$

post $club' = \mu (club, pilots \mapsto pilots(club) \cup \{mk - Pilots (n, a, l, q, [], yes)\})$

Exemple 2-19 : VDM - Définition d'une opération

Dans cette opération (voir Exemple 2-19), nous n'avons que des paramètres en entrée, qui concernent le pilote. La ligne "ext wr" indique que le club (état du système) va être modifié.
La précondition stipule que le nouveau pilote ne doit pas encore appartenir à l'ensemble des pilotes, tandis que la postcondition nous assure que le nouveau pilote a été inséré dans cet ensemble (la fonction μ permet de modifier les champs d'un objet composé, et *mk* est le constructeur qui permet de créer un nouveau pilote).

ListFlightsPilots (d1, d2 : dates)

lres : c1 :: n : RegNums

f : c2 :: r : RegNums,

dt, dl : Dates,

tt, tl : Times,

in : Names.

ext *rd club : Systems*

pre $d1 \prec_{Dates} d2$

post $lres = \{mk - c2(plane(fl), dt(fl), dl(fl), tt(fl), in(fl)) \mid fl \in flights(club) \wedge$
$pilot(fl) - p \wedge ((dt \prec_{Dates} d1 \wedge d2 \prec_{Dates} dl) \vee$
$(d1 \prec_{Dates} dt \wedge dt \prec_{Dates} d2) \vee d1 \prec_{Dates} dl \wedge dl \prec_{Dates} D2))\}\}$

Exemple 2-20 : VDM - Définition d'une opération

Cet exemple diffère quelque peu de la première opération (voir Exemple 2-19). En effet, nous y trouvons un paramètre en sortie (*lres*). L'état du système n'est pas modifié, comme l'indique la ligne "ext *rd*" ; on n'accède au *club* qu'en lecture. Une dernière différence se situe au niveau de l'utilisation d'une relation d'ordre total $\prec_{Dates}$.

2.3.6.2 Recherche des concepts

Nous avons différents éléments que nous devons pouvoir représenter dans le repository. Tout d'abord, nous devons pouvoir représenter les variables du système. Nous avons donc besoin de trouver un mécanisme nous permettant de représenter une structure arborescente de données.

Nous devons également pouvoir représenter la notion d'opération. Il faut de plus pouvoir associer à chaque opération ses arguments et ses résultats, ainsi que ses pré et post-conditions, et indiquer si l'état du système est modifié ou non.

2.3.7 Oblog

2.3.7.1 Présentation[5]

Dans cette partie, nous allons tout d'abord exposer quelques caractéristiques du langage OBLOG et nous enchaînerons par une présentation des objets du langage.

Le langage OBLOG possède les caractéristiques suivantes :

- un haut niveau d'abstraction permettant son utilisation dès les premières phases du développement d'applications ;
- une facilité d'utilisation le rendant accessible à des non spécialistes ;
- de grandes possibilités d'applications ;
- une grande expressivité permettant son utilisation dans les dernières phases du développement ;
- la possibilité de génération automatique ou semi-automatique d'applications à partir des spécifications ;
- convient bien au développement d'applications de grande taille.

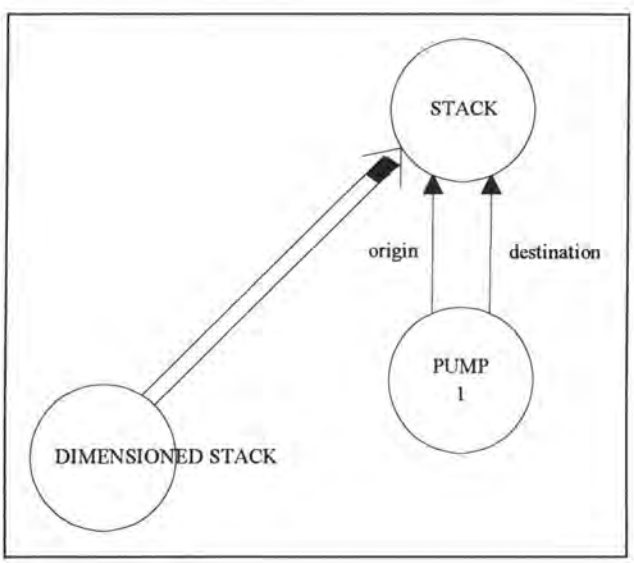
[5] La présentation de ce langage est tirée de [SER94].

De plus, nous pouvons citer certains aspects plus techniques comme :

- un langage orienté-objet ;
- un langage formel doté d'une sémantique mathématique précise et pour lequel il est possible de développer un calcul logique de vérification de la cohérence des spécifications ;
- l'inclusion de constructeurs de structures de données ;
- l'adoption du paradigme de concurrence ;
- une description uniforme des objets passifs (données) et actifs (traitements) ;
- l'existence de constructeurs permettant la construction progressive d'objets à partir des spécifications ;
- la possibilité de créer des librairies de spécification d'objets ainsi que des communautés d'objets ;
- l'inclusion de mécanismes de spécialisation ;
- l'intégration de la notion d'unités de management dans les primitives du langage.

Le langage OBLOG possède une représentation graphique. Il est constitué de différents schémas : le schéma de communauté, le schéma de définition, le schéma de comportement et le schéma d'initialisation et de modification.

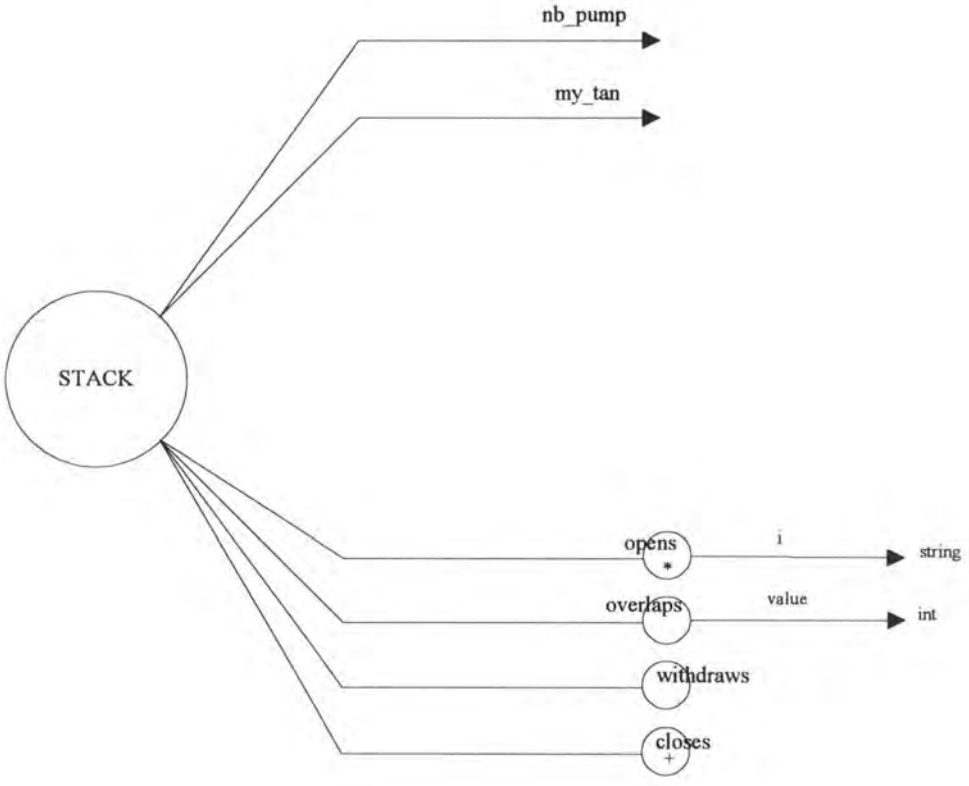
a) Le schéma de communauté



Exemple 2-21 : Oblog - Schéma de communauté

Ce schéma (voir Exemple 2-21) représente les différents objets ainsi que les relations qui existent entre eux. Dans cet exemple, la double relation entre STACK et PUMP se reflète dans les attributs *origin* et *destination* de la classe PUMP tandis que les doubles flèches représentent une relation is-a entre les objets.

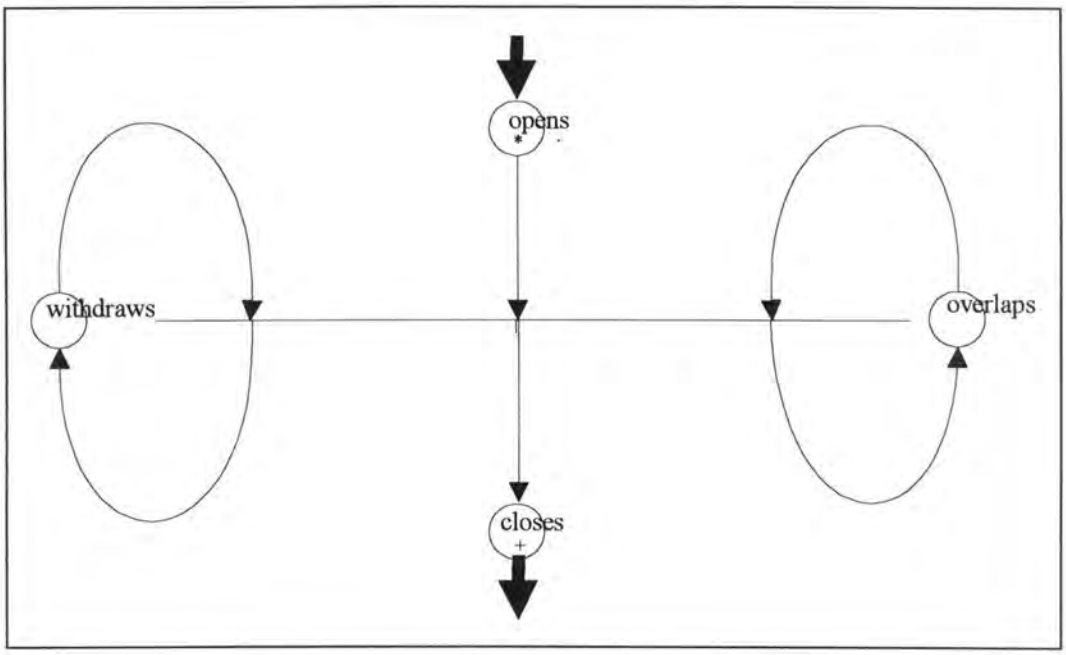
b) Le schéma de définition



Exemple 2-22 : Oblog - Schéma de définition

Ce schéma (voir Exemple 2-22) décrit les attributs et les traitements d'un objet. Les attributs de l'objet sont notés par des flèches au-dessus de l'objet, tandis que les traitements sont représentés par des cercles en bas de l'objet. Les traitements de naissance sont marqués par un * et les traitements de mort sont marqués par un +.

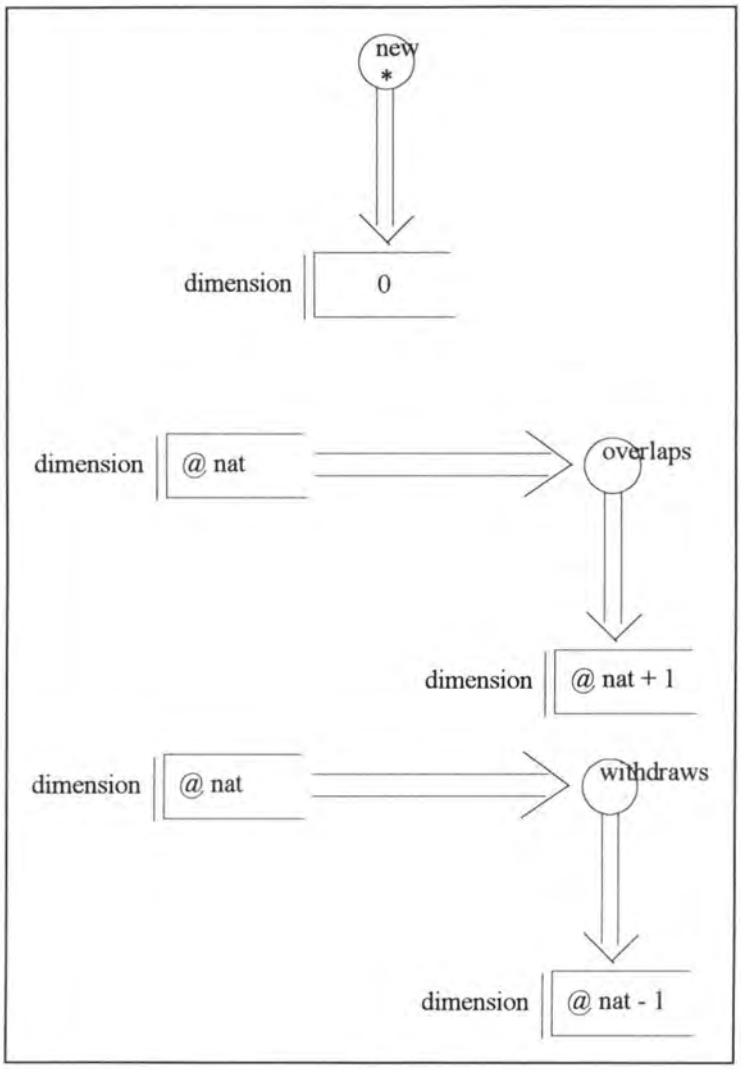
c) Le schéma de comportement



Exemple 2-23 : Oblog - Schéma de comportement

Dans ce schéma (voir Exemple 2-23), les lignes horizontales représentent des situations et les cercles représentent des actions permettant de passer d'une situation à une autre. Dans cet exemple, l'action opens est l'action de naissance. Une fois cette action accomplie, les actions *withdraws*, *overlaps* et *closes* peuvent être exécutées. Si le choix se porte sur une des deux premières, à la fin de l'action l'application se retrouvera dans la même situation. Dans le cas de l'exécution de l'action *closes*, aucune autre action ne pourra plus être exécutée.

d) Le schéma d'initialisation et de modification



Exemple 2-24 : Oblog - Schéma d'initialisation et de modification

Ce schéma (voir Exemple 2-24) spécifie les initialisations et modifications que chaque action apporte à un attribut. Dans cet exemple, l'action new assigne la valeur 0 à l'attribut dimension, l'action overlaps incrémente ce même attribut de 1, tandis que l'action withdraws le décrémente de 1.

2.3.7.2 Recherche des concepts

a) Schéma de définition

Ce schéma (voir Exemple 2-22) regroupe des définitions d'attributs et de traitements.

Pour représenter les attributs, nous devons disposer d'objets représentant une structure de données.

Afin de permettre la représentation des traitements, nous avons besoin d'objets définissant des traitements.

b) Schéma de comportement

Dans ce schéma (voir Exemple 2-23), on spécifie les successions possibles d'actions ainsi que leurs conditions d'activation. Nous avons donc besoin, pour représenter ce schéma, d'objets définissant des relations entre traitements ainsi que des conditions d'activation de ces traitements.

c) Schéma d'initialisation et de modification

Ce schéma (voir Exemple 2-24) se comporte comme un ensemble d'instructions d'affectation inscrites dans une procédure. Ces instructions doivent évidemment être également représentées. Il nous faut donc disposer d'objets nous permettant de spécifier des instructions.

2.3.8 Le langage C

2.3.8.1 Présentation

Le langage C est un langage de programmation très répandu. Parmi les raisons de son succès, nous pouvons citer notamment [BYR??] :

- la souplesse de C, langage de programmation structuré de haut niveau ;

  Les instructions du langage C sont composées de termes ressemblant à des expressions algébriques, et de mots-clés réservés tirés de l'anglais.

- certaines fonctionnalités de bas niveau, offertes généralement par les langages machine ou les langages d'assemblage ;

  C'est ce qui différencie C d'autres langages tels Pascal ou Fortran. Cette possibilité de programmation de bas niveau permet de combler le fossé entre le langage machine et les langages de haut niveau conventionnels. Cette flexibilité justifie l'emploi de C pour la programmation système comme pour des programmes applicatifs.

- la compacité des programmes objets résultant de la compilation de sources en C, ainsi que leur grande efficacité ;

  Cette aptitude est en grande partie due aux nombreux opérateurs inclus dans le langage. Son jeu d'instructions est relativement limité, tandis que les différentes versions disponibles offrent un vaste ensemble de bibliothèques de fonctions qui complètent les instructions de base. De plus, ce langage favorise le développement par les utilisateurs de leurs propres bibliothèques de fonctions, dont l'emploi permet de personnaliser l'utilisation du langage.

- la disponibilité de C sur une vaste gamme de machines, allant des micro-ordinateurs aux gros ordinateurs en passant par les minis ;

- l'indépendance de C par rapport aux systèmes d'exploitation, garantissant une grande portabilité des programmes développés dans ce langage.

  Cette portabilité est assurée par l'emploi de bibliothèques dans lesquelles sont reléguées les fonctionnalités liées à la machine. Chaque compilateur est ainsi livré avec les

bibliothèques de fonctions dépendant de la machine. Celles-ci sont relativement standardisées, et accessibles de la même façon d'un compilateur à l'autre.

2.3.8.2 Recherche des concepts

En examinant le langage C, nous constatons qu'il existe différents objets : programmes, fonctions, instructions, ... Nous avons besoin de définir un schéma permettant de représenter tout cela.

Ces différents objets ont des relations entre eux (appel, décomposition, ...) que nous devons donc également pouvoir représenter. Pour cela, nous devons disposer d'un concept de relation entre traitements.

Enfin, nous remarquons également que les programmes C utilisent des structures de données. Ces structures de données doivent être également représentées et, de plus, il nous faut spécifier qu'une donnée est un paramètre d'un traitement.

2.3.9 Le langage C++

2.3.9.1 Présentation[6]

Le langage C++ consiste en une extension du langage C lui permettant de supporter la programmation orientée objet. Pour cela, C++ dispose d'objets et de mécanismes qui étaient inconnus du langage C.

Le premier objet propre au langage C++ est la classe. Une classe est un objet contenant dans une même définition les attributs propres à cet objet et les méthodes qui lui sont attribuées.

[6] Les éléments de cette présentation sont tirés de [ACH93].

L'héritage entre objets est un mécanisme classique en programmation orientée objet. Cela signifie que l'objet dérivé hérite de tous les attributs et méthodes de son (ses) parent(s).

Nous pouvons également citer un dernier mécanisme supporté par C++. Il s'agit de la surcharge (ou overload). Une fonction ou un opérateur est dit surchargé si cette fonction ou cet opérateur possède plusieurs définitions en fonction du type de ses arguments. C'est au moment de l'exécution que le choix de la version correcte de la fonction ou de l'opérateur sera effectué.

2.3.9.2 Recherche des concepts

Nous sommes confrontés ici à un objet composite : la classe. En effet, nous sommes face à un objet comprenant, dans une même déclaration, une définition de données et une définition de traitements. Nous devons trouver un mécanisme nous permettant de représenter ces deux concepts et, de plus, nous devons pouvoir représenter les relations qui existent entre deux traitements, entre deux données ou entre un traitement et une donnée.

2.3.10 Pascal

2.3.10.1 Présentation

La langage Pascal est l'exemple type de ce que l'on appelle la programmation structurée. En effet, de par son caractère très structuré et fortement typé, il force le programmeur à développer ses programmes de façon rigoureuse. Cette rigueur conduit à une meilleure lisibilité des programmes ainsi qu'à une plus grande souplesse lors de modifications ultérieures. Il constitue en ce sens un excellent langage pédagogique. Cependant il souffre, vis-à-vis du langage C notamment, d'une certaine rigidité.

2.3.10.2 Recherche des concepts

Les concepts présents dans le langage Pascal sont identiques à ceux du langage C.

2.3.11 PML

2.3.11.1 Présentation

a) Caractéristiques du langage

Le langage PML est un langage destiné à modéliser des processus organisationnels. Chaque programme écrit dans ce langage possède trois caractéristiques [ICL??]. Il est :

- actif ;
- ouvert ;
- évolutif.

Une application PML est active en ce sens qu'elle guide l'utilisateur dans le processus. En effet, le programme informe l'utilisateur des actions possibles à un moment donné du programme (l'ensemble de ces actions évolue au cours du temps) et ne se contente pas de refuser l'exécution des actions indisponibles.

Un programme PML est ouvert car il est possible d'y intégrer des applications écrites dans un autre langage. Pour chaque application externe, le programme PML va l'invoquer, lui fournir les données et ensuite extraire les résultats.

Un programme PML est également évolutif car il est possible au programmeur de faire des modifications sans avoir à recompiler l'application entière.

b) Structure d'un programme PML [ICL??]

La structure principale d'un programme PML est le rôle. Un rôle est un objet possédant son propre espace de données et sa propre étendue d'exécution. Les différentes activités d'un processus peuvent être représentées par des rôles. Des rôles peuvent être créés au cours de l'exécution de l'application lorsque de nouvelles activités débutent. A la fin de ces activités, les rôles se terminent. Il est aussi possible d'échanger de l'information entre rôles via des

interactions. Une interaction est un canal de communication permettant à un rôle de copier le contenu de variables dans l'espace de stockage d'un autre rôle.

A chaque personne impliquée dans le processus est associé un agent. Lorsqu'une activité doit être assignée à un agent, le rôle correspondant lui est attribué et est affiché sur sa station de travail.

c) Construction d'un programme PML [ICL??]

Une des particularités fondamentales de PML est la procédure de construction de programmes. Plutôt que de fournir un compilateur indépendant, la compilation et le chargement apparaissent dans l'interface PML. Tous les modèles débutent de la même façon : un rôle et un agent (appelé root). La conception consiste en la modification de ce rôle et la création d'autres agents.

d) Les objets du langage [ICL??]

Il existe beaucoup d'objets différents dans ce langage. Ils peuvent aller du plus simple (un booleen, par exemple) à un objet très structuré (un rôle). Nous allons ici décrire les objets principaux existant en PML.

Les entités

Les entités constituent les données sur lesquelles les rôles vont agir. La définition d'une entité est, en fait, la définition d'un nouveau type de données. Le nom de la classe ainsi définie est également le nom du type introduit.

PML propose quatre classes d'entités primitives. Il s'agit des classes Bool, Int, Real et String. PML définit également une classe Entity qui pourra être utilisée comme sur-type afin de créer d'autres entités.

La définition d'entités se fait de la façon suivante :

entity-class ::= classname « isa » classname
[« with » [« parts » { data-decl }] « end » « with »]

data-decl ::= propname { « , » propname } « : » type [« := » expression]

type ::= { constructor } classname

La nouvelle entité est donc définie comme un sous-type d'une entité existante (voir Exemple 2-25). Cette nouvelle entité hérite de toutes les propriétés de son sur-type. De plus, le sous-type peut redéfinir des propriétés de son sur-type.

Exemple [PML94]

```
Employé isa Entity with
parts
        nom : String := ''
        numéro : Int := 5192
        département : String
end with
```

Exemple 2-25 : PML - Définition d'une entité

Les actions

Une définition d'action est analogue à une définition de procédure dans un langage de programmation standard. Elle définit les opérations que les rôles auront à accomplir lors de leur exécution.

Les actions possèdent les attributs suivants :

- *in* et *out* qui définissent les paramètres de l'action ;

- *resources* définissant les variables locales connues uniquement de l'action ;
- *parts* qui contient les commandes à exécuter ;
- *preconds* définissant les préconditions devant être respectées lors de l'appel de l'action ;
- *postconds* définissant les postconditions de l'action ;
- *termconds* définissant les conditions qui déterminent la fin de l'action.

Ici aussi, il existe un mécanisme d'héritage. L'action sous-type hérite des *in*, *out*, *resources* et *parts* de son action sur-type.

La définition d'actions se fait de la façon suivante :

```
action-class   ::=   classname « isa » classname
                     [ « with »
                           [ « in » { data-decl } ]
                           [ « out » { out-decl } ]
                           [ « resources » { data-decl } ]
                           [ « parts » { exec-decl } ]
                           [ « preconds » [ expression ] ]
                           [ « postconds » [ expression ] ]
                           [ « termconds » [ expression ] ]
                     « end » « with » ]

data-decl      ::=   prop-name { « , » prop-name } « : » type [ « := » expression ]

out-decl       ::=   prop-name { « , » prop-name } « : » type

exec-decl      ::=   prop-name « : » command [ « when » expression ]

command        ::=   action-call
                   | assignment
                   | conditional
                   | iteration
```

| « { » command { « ; » command } « } »

Les rôles

Le rôle est le concept central de PML. C'est un objet actif qui encapsule ses données et son comportement.

Les rôles peuvent avoir les attributs suivants :

- *classes* contenant une liste de classes connues par le rôle ;
- *resources* définissant les variables locales au rôle ;
- *actions* définissant le comportement du rôle. Cet attribut agit de la même façon que l'attribut *parts* des actions ;
- *initially* est une proposition portant sur les *resources* et définissant les hypothèses que le rôle fait sur l'état initial de ces variables ;
- *always* définissant des propriétés devant être respectées à tout moment par les *resources* du rôle ;
- *termconds* déterminant la condition de terminaison du rôle.

Comme les autres objets du langage, les rôles héritent des propriétés de leur sur-type. Ils peuvent cependant les redéfinir, mais il est à noter que si un rôle redéfinit les propriétés *initially* ou *always*, les nouvelles conditions seront constituées par la conjonction des conditions du sur-type et des conditions du sous-type. De même, si un rôle redéfinit la propriété *termconds*, le rôle se terminera si la condition du sur-type ou la condition du sous-type est vérifiée.

Les interactions

Les variables définies à l'intérieur d'un rôle sont connues de ce seul rôle. Si des communications de données entre deux rôles sont nécessaires, il faut alors créer un canal de communication entre les rôles. Ce canal est appelé une interaction.

La création d'une interaction se fait de la façon suivante [PML94] :

```
tp : takeport Entity
gp : giveport Entity
```

Les deux variables représentent les deux extrémités du canal de communication. La première permet de mettre une donnée du type spécifié (ou d'un de ses sous-types) à la disposition d'un autre rôle et la deuxième permet au rôle auquel est destinée la donnée d'y accéder.

```
NewInteraction (giver = gp, taker = tp)
```

Cette instruction crée une interaction, c'est-à-dire elle crée un canal de communication entre les variables *gp* et *tp*.

Ensuite, il existe deux instructions d'échange de données. Il s'agit des instructions *give* et *take* servant respectivement à placer une donnée dans le canal et à prendre cette donnée. Elles s'utilisent comme suit :

```
int : Int := 5
Give (interaction = gp, gram = int)

entity : Entity
int : Int
Take (interaction = tp, gram = entity) ;
int := entity as Int ;
```

Les agents

Dans un processus organisationnel, les personnes jouent un rôle central. Dès lors, il est essentiel de se concentrer sur tout ce qui peut leur permettre de travailler plus efficacement.

C'est la raison d'être de la notion d'agent en PML. Il est possible de créer des utilisateurs et de leur affecter des rôles. A ce moment, tout utilisateur verra sur l'écran de sa station de travail tous les rôles lui étant destinés et eux seuls. L'utilisateur a donc à sa disposition toute l'information qui lui est utile.

Tout agent possède un username et un mot de passe.

e) L'action BehaveAs [ICL??]

Les processus organisationnels évoluent. Il faut donc que les modèles informatiques que l'on construit soient également évolutifs. L'action BehaveAs répond à cette demande en permettant de changer le comportement d'un rôle lors de l'exécution du modèle.

Lors de l'appel de l'instruction BehaveAs, l'instance du rôle que l'on veut modifier ainsi que la modification elle-même doivent être fournies. Le compilateur se sert alors de sa propriété d'évolution décrite au début de ce chapitre pour compiler le nouveau comportement du rôle sans devoir stopper le modèle et le recharger ensuite. Si la compilation se passe sans erreur, le modèle continue son exécution tandis que dans le cas d'erreur de compilation, un string d'erreur est renvoyé en sortie par l'action BehaveAs.

Il est à noter que, bien que cette action soit très puissante en raison de la possibilité qu'elle a de modifier le comportement d'un rôle au moment de l'exécution, son utilisation abusive rend les programmes complexes et difficiles à comprendre.

Une utilisation un peu différente de cette instruction peut être faite pour simuler des variables globales. En effet, chaque variable appartient à un et un seul rôle et est inconnue de tous les autres. Cependant, il est parfois utile de travailler avec des variables globales. De telles variables n'existent pas en PML. Pour les simuler, on peut utiliser l'instruction BehaveAs. Il suffit que chaque rôle ait une variable propre et que l'un d'entre eux transfère la valeur de sa variable dans les variables correspondantes. En prévoyant une action se déclenchant chaque fois qu'une valeur est stockée dans cette variable, et dont le but est de transférer sa valeur aux

variables des autres rôles, toutes les variables correspondantes auront, à tout moment, la même valeur.

2.3.11.2 Recherche des concepts

a) Les entités

Les entités PML ont pour but de stocker de l'information. Nous devons pouvoir représenter une telle structure de données : sa structure arborescente, ses éléments et leur type.

b) Les actions

Les actions PML sont analogues à des procédures dans les langages de programmation traditionnels. Il nous faut donc représenter la notion d'opération prenant des paramètres en entrée et produisant des résultats.

c) Les rôles

Un rôle est un objet regroupant dans une même définition des structures de données et des traitements. La partie de la définition reprenant les structures de données sera représentée de façon semblable aux entités. Afin de modéliser les traitements, nous utiliserons les mêmes structures que pour les actions. Le lien entre les deux parties de la définition devra également être spécifié.

d) Les interactions

Une interaction permet à deux rôles de communiquer. Il s'agit donc d'un objet permettant de créer des relations entre deux rôles. Nous pouvons dès lors la considérer comme une relation entre deux objets, à savoir, les rôles entre lesquels est définie l'interaction.

e) Les agents

Un agent est un utilisateur du système. Nous devons disposer, pour le modéliser, d'une structure nous disant à qui appartient tel traitement ou telle donnée.

3. Création du méta-schéma

3.1 Introduction

La section suivante de ce chapitre est destinée à la synthèse des concepts rencontrés lors du chapitre précédent. Cette synthèse va nous permettre de distinguer deux grandes parties parmi ces concepts : l'une concernant les données et l'autre les traitements. Pour ce qui est des données, un méta-schéma existe déjà au sein de l'outil CASE DB-Main.

La troisième section est donc consacrée à la présentation de celui-ci. Nous pourrons ensuite faire une critique de ce schéma, en tenant compte des concepts repérés. Cela fera l'objet de la quatrième section.

Enfin, la dernière section présentera les modifications que nous proposons au schéma concernant les données, et l'extension que nous envisageons pour modéliser la partie traitements d'une application.

3.2 *Synthèse des concepts*

Parmi les différents modèles que nous avons étudiés, nous pouvons remarquer que certains concepts reviennent systématiquement.

En effet, nous retrouvons toujours le concept de données. Il s'agit d'objets destinés à stocker de l'information ou à permettre à cette information de circuler entre différents traitements. Ces données sont toujours organisées en structure arborescente. Il nous faudra pouvoir les représenter. Pour cela, un schéma de repository a déjà été créé dans l'outil DB-Main. Ce schéma sera expliqué dans la section suivante.

Ensuite, nous voyons le concept de traitement. Celui-ci est présent sous différentes formes, à savoir un programme, une fonction, une méthode, une spécification, ... Quoi qu'il en soit, tous ces traitements ne se distinguent que par un niveau d'abstraction différent et peuvent tous se représenter de la même façon.

Ces traitements interagissent les uns avec les autres. Nous pouvons citer comme exemple une relation entre un programme et ses fonctions, entre une spécification et ses parties pré et post-conditions, ...

Ils possèdent également des liens vers des paramètres. Cette notion de paramètre peut être très vaste. Elle englobe aussi bien la notion de variable, que celle de document, de message, ...
Il nous faudra aussi spécifier le rôle joué par le paramètre au sein d'un traitement. Il peut en effet constituer un input ou un output, n'intervenir que si une certaine condition est satisfaite, ...
Dans le cas où il s'agit d'une variable, une structure doit être prévue pour prendre en compte la valeur de celle-ci.

Une notion importante qu'il va nous falloir gérer est celle de classe. En effet, une classe consiste en la définition, au sein d'un même objet, de données et de traitements.

De plus la notion de classe est toujours accompagnée de mécanismes tels que la spécialisation, l'instanciation, ... Un lien particulier se crée alors entre les objets sur lesquels portent ces mécanismes.

Un autre concept que nous avons rencontré est celui d'acteur. En effet, certains modèles spécifient l'appartenance d'un traitement ou d'une donnée à une personne ou à un groupe de personnes.

Un point qui a également été souligné lors de notre analyse concerne le type des données. Plusieurs modèles ne se contentent pas des types standards. Souvent, nous avons besoin de traiter des structures comme des ensembles, des listes, des séquences, ... voir des types encore plus particuliers comme un *giveport* dans le cas du langage PML.

3.3 *Présentation du méta-schéma de DB-Main*[7]

L'élément principal du repository est l'entité *System* (voir Figure 3-1). Le repository ne contient qu'un seul système.

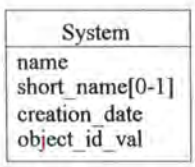


Figure 3-1 : Entité System

L'entité *System* est caractérisée par un nom, un nom court ainsi qu'une date de création. Le dernier attribut n'est pas important dans le cadre de ce mémoire.

Le système est relié à tous les schémas ainsi qu'à tous les documents du repository (voir Figure 3-2). Cela se fait au moyen de la relation *Sys_sch*.

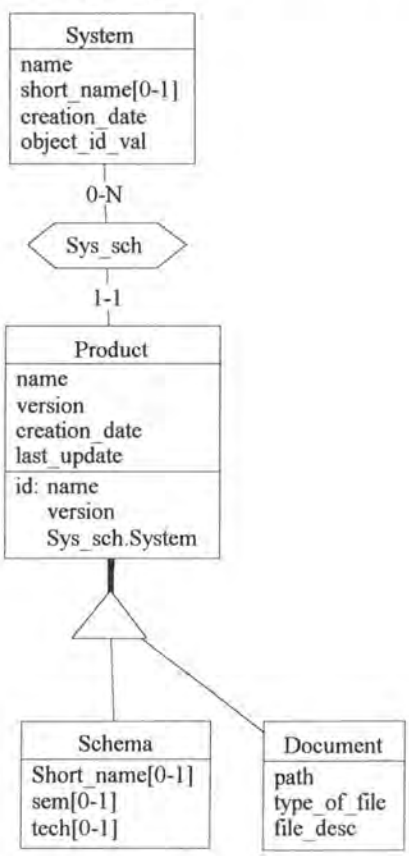


Figure 3-2 : Relation entre le system *et les* schemas

[7] La présentation du méta-schéma est extraite de [DBM94].

Aussi bien l'entité *Schema* que l'entité *Document* héritent des propriétés du *Product*.
Les attributs *sem* et *tec* du schéma servent à contenir une description sémantique et technique du schéma. Ces attributs sont bien sûr facultatifs.

Il est également possible de représenter des liens entre les différents schémas présents dans le repository. Ces liens sont représentés pas l'entité *Connection* (voir Figure 3-3).

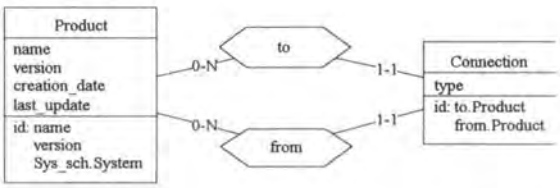


Figure 3-3 : Relations entre les products

Il est, par exemple, possible de dériver un schéma à partir d'un autre. Nous pouvons conserver cette information grâce à cette entité et aux deux relations *from* et *to*.

Un schéma est composé d'un certain nombre d'objets (voir Figure 3-4). Ces objets peuvent être soit des entités ou des relations, soit des attributs.

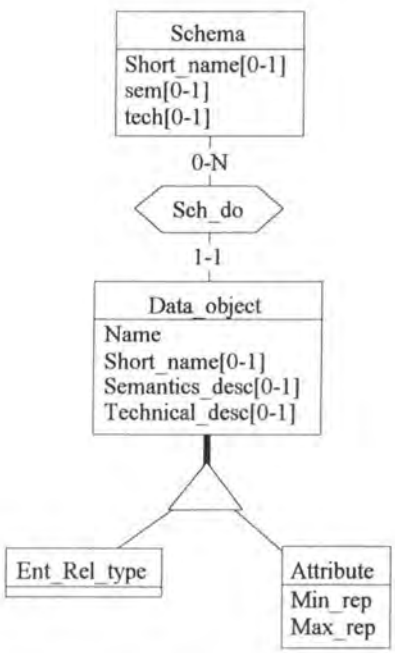


Figure 3-4 : Relation entre le schema *et ses* objects

A nouveau, aussi bien l'entité *Ent_Rel_type* que l'entité *Attribute* héritent des propriétés de *Data_object*.

Nous allons commencer à développer la partie entité-relation, avant de nous attarder sur la notion d'attributs.

L'entité *Ent_Rel_type* (voir Figure 3-5) peut être de deux types : entité (*Entity_type*) ou relation (*Rel_type*).

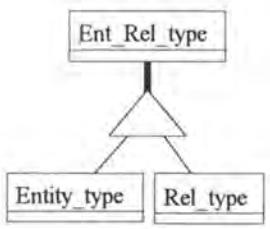


Figure 3-5 : Type d'entité et relation

Un type d'entité (*Entity_type*) se définit comme étant une classe d'objets appartenant à la réalité. Tout type d'entité, comme toute relation, est un sous-type d'un *Ent_Rel_type*. Cela signifie à nouveau qu'il hérite de l'ensemble des propriétés de celui-ci.

Nous pouvons trouver des collections de types d'entités (voir Figure 3-6). Une collection représente donc un ensemble d'entités. Cette *Collection* va être reliée au *schema* ainsi qu'à l'ensemble des entités qui la composent. Un type d'entité peut appartenir à plusieurs collections.

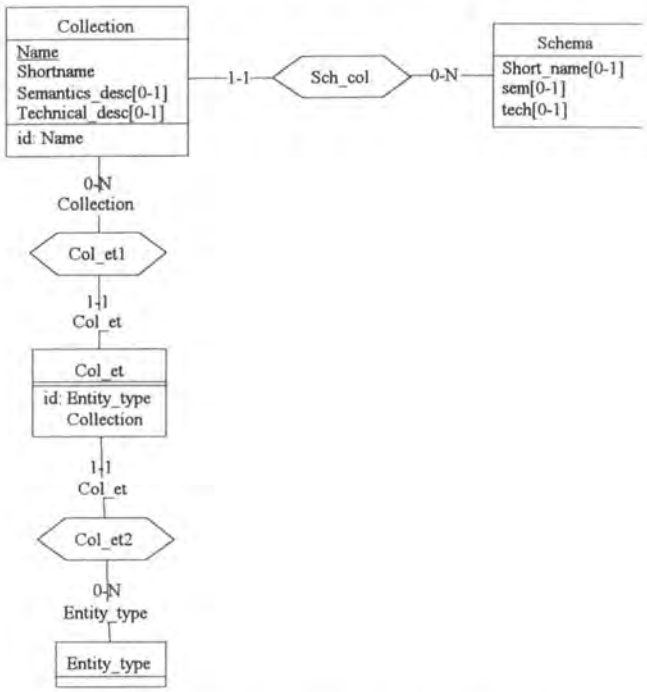


Figure 3-6 : Collection *de types d'entités*

Chaque *Collection* est identifiée par son nom. Elle possède également un nom court, une description sémantique et une description technique.

Tout type d'entité peut également participer à des relations (voir Figure 3-7). En participant à une relation, le type d'entité joue un rôle. La relation définit donc une correspondance entre deux ou plusieurs types d'entité où chacun assume un rôle donné.

Cette correspondance entre les types d'entité et leurs rôles est représentée de la manière suivante :

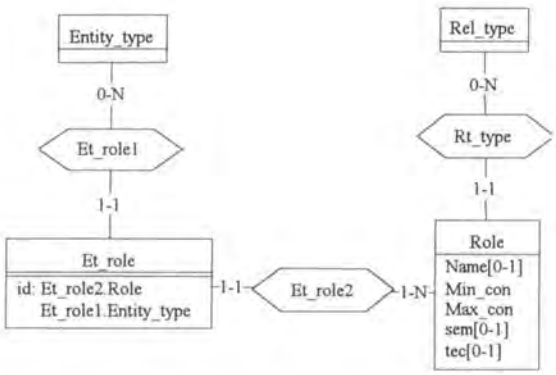


Figure 3-7 : Relation entre types d'entités

Aussi bien *Rel_type* que *Role* peuvent être joués par un ou plusieurs *Entity_type*. On doit donc indiquer le nombre minimum (*min_con*) et le nombre maximum (*max_con*) d'occurrences de ce *Rel_type* dans lesquels une même occurrence de cet *Entity_type* peut apparaître.
Le rôle possède en plus une description sémantique et une description technique.

Avant d'analyser la notion d'attributs, il nous reste à regarder quelque peu la notion de *Cluster* (voir Figure 3-8).
Cette notion permet de pouvoir représenter le processus de généralisation/spécialisation. Un *cluster* est un groupe d'entités. Un *cluster* est caractérisé par le type de contrainte envisagée. Un *cluster* possède un critère (*criterion*) qui définit le contexte sémantique de spécialisation du type d'entité générique. Cette notion correspond au triangle dans la représentation graphique.

Afin de bien comprendre ce concept, prenons un exemple : le type d'entité PERSONNE se spécialise en HOMME et FEMME. Le critère de spécialisation est le sexe. Ce critère regroupe les types d'entités spécifiques HOMME et FEMME.

L'entité *Sub_type* ne contient qu'un seul attribut (*Value*) qui représente l'instance du critère du *cluster*.

Pour notre exemple, le critère de spécialisation est le sexe. La valeur de ce critère est soit "masculin" soit "féminin".

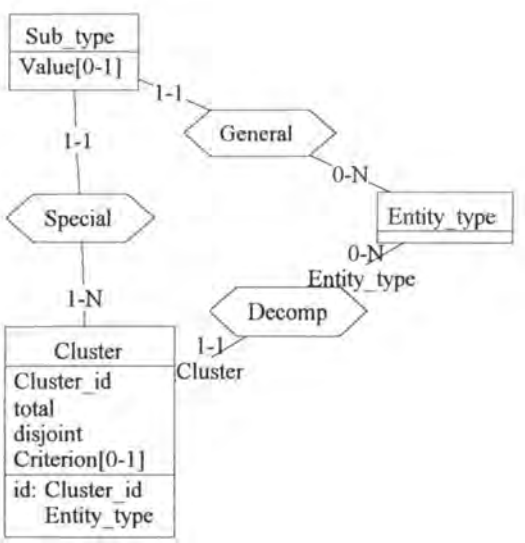


Figure 3-8 : Notion de cluster

Comme nous l'avons déjà dit tous les types d'entités et toutes les relations sont des sous-types de l'entité *Ent_Rel_type*. Grâce à cette entité et à l'entité *Owner_of_att*, ils peuvent possèder de 0 à N attributs (voir Figure 3-9).

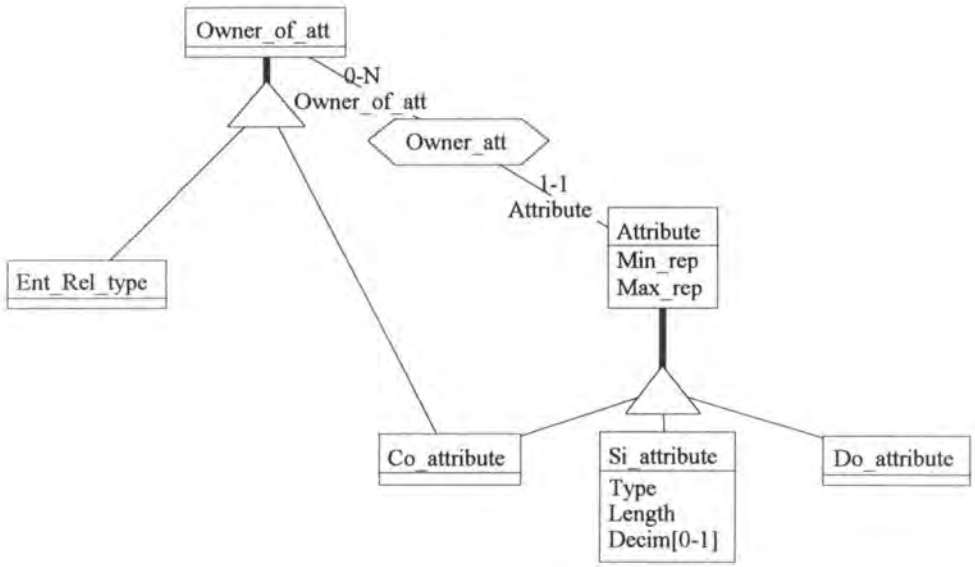


*Figure 3-9 : Notion d'*attribute

L'attribut (*Attribute*) représente une caractéristique d'un type d'entité ou d'une relation.
Un attribut peut représenter de manière non simultanée un attribut décomposable (*Co_attribute*), un attribut simple (*Si_attribute*) ou un attribut de type domaine (*Do_attribute*).
Le nombre de valeurs d'*Attribute* que l'on peut retrouver associées à un *Ent_Rel_type* ou à un *Co_attribute* doit être indiqué. Ce nombre est spécifié par une valeur minimum (*min_rep*) et une valeur maximum (*max_rep*). Si *min_rep* est nul, cela signifie que l'attribut est facultatif.

Un attribut décomposable est un attribut qui a pour composant d'autres attributs. Cela se représente au moyen de la relation *Owner_att*.
Un attribut simple est un attribut qui est de type standard. Les types standards d'un attribut sont l'alphanumérique, le numérique, la date, le booléen, l'entier et le type flottant. L'attribut simple est caractérisé par un type (*type*), une longueur (*length*) et éventuellement par des décimales (*decim*).
L'attribut est de type domaine si l'on permet de lui associer un domaine comme valeur.

Il reste maintenant à examiner le concept de groupe (voir Figure 3-10 et Figure 3-11). Un groupe (*group*) est une collection d'*attribute* et/ou de *role* et/ou de *group* attaché à une *data_object*. Il va être utilisé pour représenter des notions telles que l'identifiant primaire, identifiant secondaire, clé d'accès, ...

En effet, le groupe joue une ou plusieurs fonctions spécifiques pour ce *data_object*.

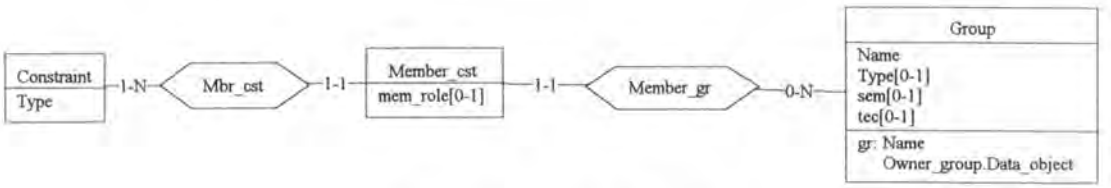


Figure 3-10 : Notion de group

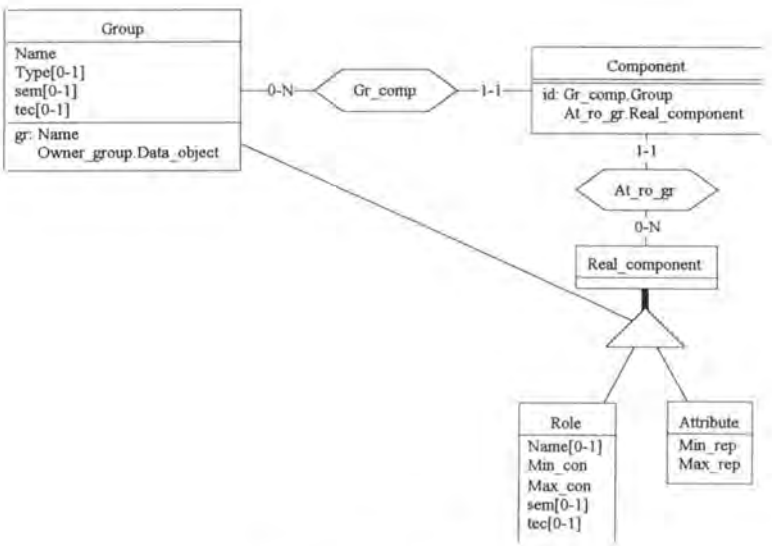


Figure 3-11 : Notion de group

Le dernier élément que nous n'avons pas encore évoqué, et qui nous sera utile par la suite, est celui de *Generic_object* (voir Figure 3-12).

Un *Generic_object* est un type d'entité dont presque tous les autres héritent. Le but de ce *Generic_object* est de disposer d'un objet générique qui puisse représenter n'importe quel autre objet du repository.

Il est identifié par un numéro (*Object_id*), un type (*object_type*) qui indique quel est le type d'entité qui hérite de cette occurrence du *generic_object*. Trois autres attributs sont présents pour indiquer la position de l'objet dans la fenêtre (*posx* et *posy*), ainsi que la couleur de l'objet (*color*).

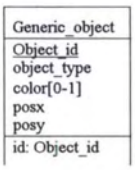


Figure 3-12 : Generic_object

La page suivante contient une vue complète de ce méta-schéma.

db_main/1

Generic_object
Object_id
object_type
color[0-1]
posx
posy
id: Object_id

System*
name
short_name[0-1]
creation_date
object_id_val

0-N
Sys_sch
1-1

Product*
name
version
creation_date
last_update
id: name
version
Sys_sch.System*

0-N
to
1-1
Connection*
type
id: to.Product*
from.Product*
0-N
from
1-1

Schema
Short_name[0-1]
sem[0-1]
tech[0-1]

Document
path
type_of_file
file_desc

Sch_col
0-N
1-1
Collection*
Name
Short_name
Semantics_desc[0-1]
Technical_desc[0-1]
id: Name
0-N
Collection
Col_et1
1-1
Col_et
Col_et*
id: Col_et2.Entity_type
Col_et1.Collection
1-1
Col_et
Col_et2
0-N
Entity_type

0-N
Sch_do
1-1
Data_object*
Name
Short_name[0-1]
Semantics_desc[0-1]
Technical_desc[0-1]
0-N

Owner_of_att
0-N
Owner_of_att
Owner_att
1-1
Attribute
Attribute
Min_rep
Max_rep

Sub_type*
Value[0-1]
1-1
Special
1-N
1-1
General
0-N

Ent_Rel_type

Cluster*
Cluster_id
total
disjoint
Criterion[0-1]
id: Cluster_id
Decomp.Entity_type
-1
ister
Decomp
0-N
Entity_type
Entity_type
0-N
Et_role1
1-1
Et_role*
id: Et_role2.Role*
Et_role1.Entity_type
1-1
Et_role2
1-N
Role*
Name[0-1]
Min_con
Max_con
sem[0-1]
tech[0-1]

Rel_type
0-N
Rt_type
1-1

Co_attribute

Si_attribute
Type
Length
Decim[0-1]

Do_attribute

Real_component
0-N
At_ro_gr
1-1
Component*
id: Gr_comp.Group
At_ro_gr.Real_component

Group
Name
Type[0-1]
function
sem[0-1]
tech[0-1]
gr: Name
Owner_group.Data_object*
0-N
Gr_comp
1-1
Owner_group

Constraint*
Type
1-N
Mbr_cst
1-1
Member_cst*
mem_role[0-1]
1-1
Member_gr
0-N

3.4 *Critique du repository de DBMain*

A l'heure actuelle, le repository de DB-Main représente les données persistantes. Pour représenter les traitements, nous avons besoin de décrire dans le repository les données non persistantes (à savoir les variables, les documents, les messages). Pour ce faire, nous proposons de créer un deuxième schéma qui contiendra ces données non persistantes. Dans le cas de modèles du style du modèle des traitements de Merise (où des messages circulent entre différents traitements), nous proposons de créer également un schéma de données, différent des schémas des données persistantes et non persistantes, permettant de représenter ces messages. Les relations entre ces différents schémas seront matérialisées au moyen de l'entité *connection*.
Nous n'avons pas trouvé utile de faire une distinction entre la notion de message et celle d'évènement. De même, nous avons préféré utiliser la structure d'un schéma de données pour modéliser ces notions plutôt que d'en créer une nouvelle. En effet, toutes les structures nécessaires sont présentes dans ce schéma.

Cependant aucune structure n'existe, pour le moment, pour représenter un traitement. Cela constitue la majeure partie de notre travail.

D'autres problèmes vont être rencontrés. En effet, seuls les types standards peuvent être utilisés. De plus, il n'est pas possible de définir ses propres types.

3.5 *Proposition d'extension*

Le schéma que nous proposons ci-dessous permet la représentation des principaux concepts cités lors du point relatif à la synthèse des concepts. Cependant, tous les problèmes ne peuvent être règlés sans apporter certaines modifications au schéma du repository existant.
Nous allons débuter en exposant les quelques modifications qui nous paraissent indispensables, avant de présenter notre extension.

Modifications de la partie données

Comme nous l'avons dit dans la synthèse des concepts des différents modèles, il est nécessaire de pouvoir donner une valeur à une variable. Cette variable est représentée par une *Entity-type* ayant un attribut unique de même nom.
Le repository ne permettant pas de stocker cette information, il nous semble utile d'y apporter une modification. Il suffirait en effet d'ajouter un attribut *Value* de type string à l'entité *Si_Attribute*.

Un autre problème plus important est lié à la notion de type. Nous ne pouvons pas représenter des types non standards comme les ensembles, les séquences, ... Les deux attributs *Min_rep* et *Max_rep* de l'entité *Attribute* nous permettent de spécifier la notion générale d'ensemble, mais ne permet aucune distinction entre l'ensemble, la séquence, ...
La solution à ce problème consisterait à ajouter à l'entité *Attribute* un attribut qui permettrait d'indiquer cette distinction.

Le type d'un attribut simple devrait également être élargi. En effet, actuellement, les seuls types qui peuvent être représentés sont les entiers, les réels, les caractères, les chaînes de caractères, les booléens, les dates et les attributs décomposables. Or, certains modèles que nous avons étudiés requièrent d'autres types particuliers. Le langage PML, par exemple, nécessite la représentation du type *giveport*.
Il faudrait donc pouvoir élargir la liste des types possibles. Cependant, tous les types ne peuvent bien sûr pas être envisagés. Une solution consisterait à indiquer dans l'attribut *Type* de

l'entité *Si_attribute* un caractère particulier indiquant que le type est non standard et stocké dans la partie *Tec* de cette entité (ex. : *type = giveport Entity*).
Une autre modification, plus importante mais non indispensable, nous semble intéressante. En effet, la plupart des modèles permettent la définition de types personnalisés (ex.: un record en Pascal). Cette notion n'est pas présente à l'heure actuelle. Il nous semble donc utile de créer un nouveau schéma qui contiendrait les représentations de ces types. Un des avantages que cette solution apporterait serait une comparaison immédiate du type de différentes variables.

Extension du repository

L'élément principal de cette partie du repository (voir Figure 3-14) est la *processing unit (PU)*. Cette notion de *processing unit* est très large. Tout traitement peut être représenté par une *processing unit*. Un programme C est une *processing unit*. Il en est de même pour une instruction C, pour une spécification pré-post, pour un data-flow diagram, ...

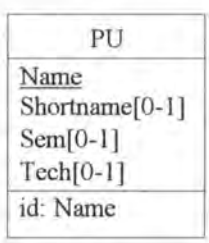


Figure 3-13 : Processing unit

Une *processing unit* (voir Figure 3-13) a un nom qui peut être le nom du programme (dans le cas où la *processing unit* représente un programme) ou un nom donné arbitrairement (une instruction, par exemple, n'a pas de nom et il nous sera nécessaire de lui en donner un arbitrairement). On peut aussi associer à une *processing unit* un *short-name* (cet attribut est facultatif).
Tout comme dans le repository existant, nous avons inclus dans l'entité *processing unit* des attributs *sem* et *tech*. L'attribut *tech* contiendra une référence au fichier source où est définie la *processing unit*.

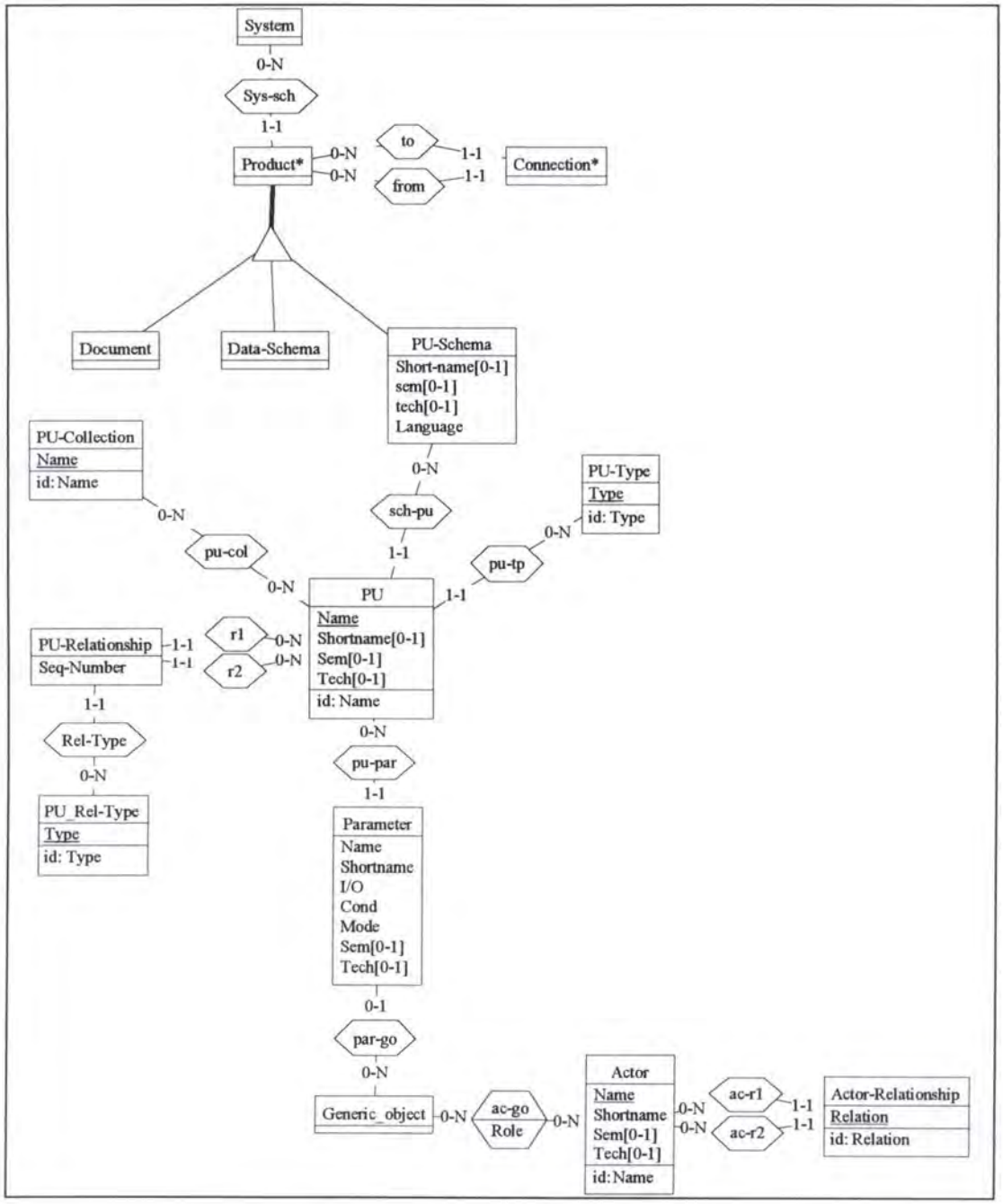


Figure 3-14 : Extension du repository

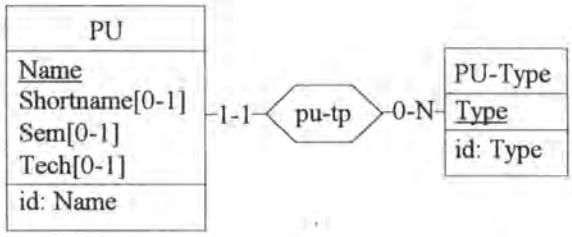


Figure 3-15 : Type d'une processing unit

Chaque *processing unit* appartient à un type déterminé (voir Figure 3-15). C'est ce type qui sera stocké dans l'entité *PU-Type*. Il pourra prendre des valeurs telles que program, function, affectation, simple conditional branch, ... Cet attribut *type* aurait pu être représenté comme un attribut de *processing unit*, mais nous avons préféré en faire un type d'entité (représentation par valeur) afin de pouvoir faire plus efficacement des recherches des différentes *processing units* d'un type donné.

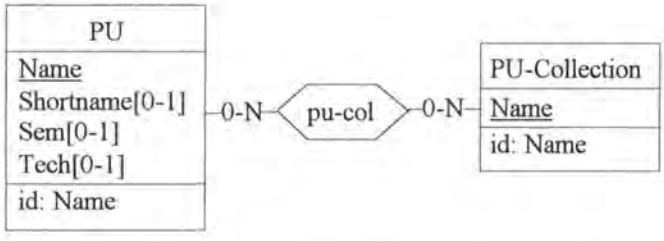


Figure 3-16 : Collection de processing units

Les *processing units* peuvent être regroupées en *PU-Collection* (voir Figure 3-16). Une *PU-Collection* est un ensemble de *processing units*. Une *PU-Collection* contiendra un nom (qui sera déterminé arbitrairement ou par l'utilisateur).

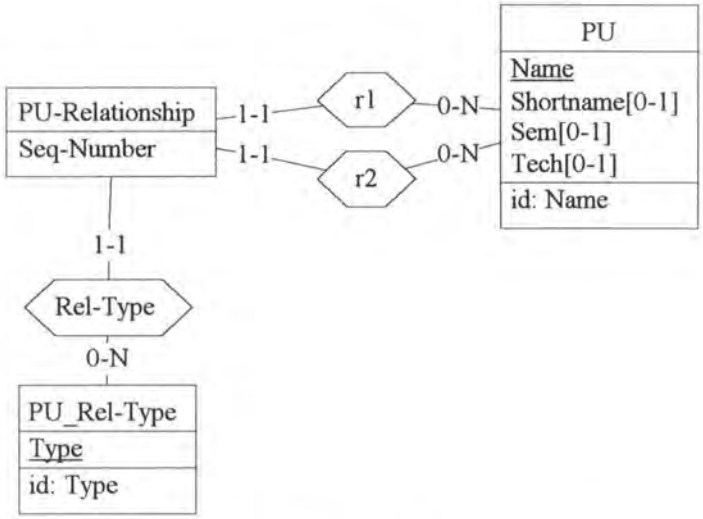


Figure 3-17 : Relation entre processing units

Entre les *processing units* peuvent exister des relations (voir Figure 3-17). Elles seront matérialisées par l'entité *PU-Relationship*. Le type de la relation sera indiqué par l'attribut *type* de l'entité *PU_Rel-Type*. Cette relation pourra être, par exemple, "module-of", "function-of", "instruction-of", ...

L'attribut *seq-number* va nous servir à organiser en séquence les *processing units* de même niveau de décomposition. En effet, pour un programme C, nous allons créer une *processing unit* représentant le programme tout entier. Nous allons ensuite décomposer cette *processing unit* en différentes autres *processing units* représentant les différentes fonctions du programme. Ces fonctions seront, à leur tour, décomposées en instructions. Si nous observons la décomposition d'une fonction en instructions, nous réalisons que l'ordre dans lequel apparaissent les différentes instructions est primordial pour le fonctionnement du programme. Etant donné que nous tenons à pouvoir, à partir de notre représentation, recréer le programme, il nous est nécessaire de représenter cet ordre. C'est la raison d'être de cet attribut *seq-number*. Chaque relation entre *processing units* recevra un numéro permettant de retrouver l'ordre dans lequel les processing units ont été créées.

Nous avons à nouveau choisi de représenter le type de la relation au moyen d'un type d'entité, pour la même raison que dans le cas de la *processing unit*.

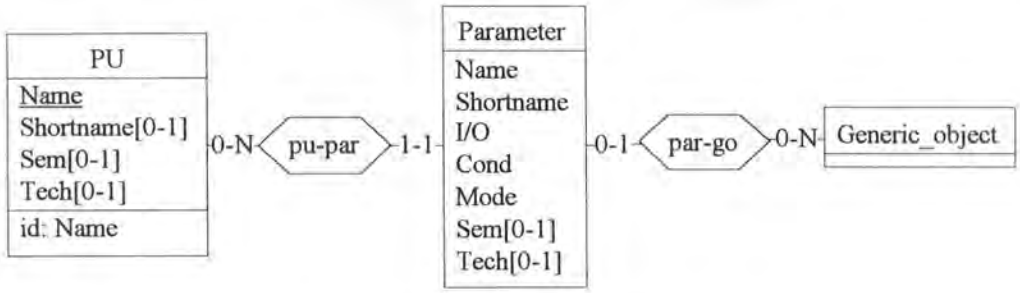


Figure 3-18 : Paramètres d'une processing units

Une *processing unit* peut avoir des paramètres (voir Figure 3-18). Par exemple, si l'on considère l'appel d'une fonction, cette fonction requiert des données et produit des résultats. Dans un diagramme des flux, par exemple, les *processing units* sont les traitements et les *parameters* sont les fichiers et les messages. L'entité *parameter* comprendra un nom (souvent le nom de la variable) ainsi qu'un *shortname*. L'attribut *I/O* indiquera si le paramètre est un paramètre en entrée ou en sortie, tandis que l'attribut *Cond* indiquera la condition d'existance du paramètre. L'attribut *mode* servira à préciser si un paramètre de fonction a été passé par valeur ou par adresse. On retrouve également les attributs *Sem* et *Tech*. Une référence à l'endroit du fichier source, où a été défini le paramètre, sera conservé dans l'aatribut *Tech*.

Les différents *parameters* seront ensuite reliés au schéma des données (persistantes ou non, selon le cas).

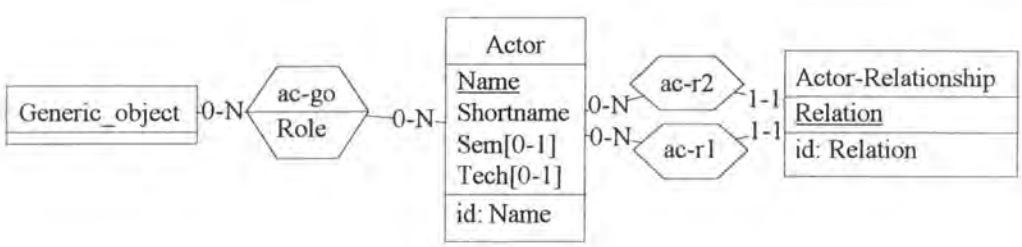


Figure 3-19 : Notion d'acteur

Dans certains modèles, il est possible d'affecter à une personne ou à un groupe de personnes un traitement ou une donnée. Ce concept sera représenté dans le repository par la notion d'acteur (voir Figure 3-19). L'acteur recevant une donnée ou effectuant un traitement sera représenté par une entité *actor* qui sera reliée au *generic object* considéré. Le premier paramètre de cette

entité définira un nom pour l'utilisateur ou le groupe d'utilisateurs et le paramètre *role* précisera le rôle de cet acteur (utilisateur, développeur, ...).

Il peut exister également des relations entre les acteurs. Par exemple, un acteur peut regrouper plusieurs personnes étant, elles-mêmes, des acteurs. C'est la raison pour laquelle nous avons ajouté l'entité *actor-relationship*.

Dans ce repository, il est aussi possible de représenter le fait que plusieurs processus peuvent s'exécuter en parallèle. Cela peut se représenter de deux manières différentes.

Soit une *processing unit* est créée pour représenter l'ensemble des processus, et à chaque processus correspond une *processing unit*. La relation entre les *processing units* individuelles et la *processing* unit générale décrira le fait que tous ces processus peuvent être exécutés en parallèle.

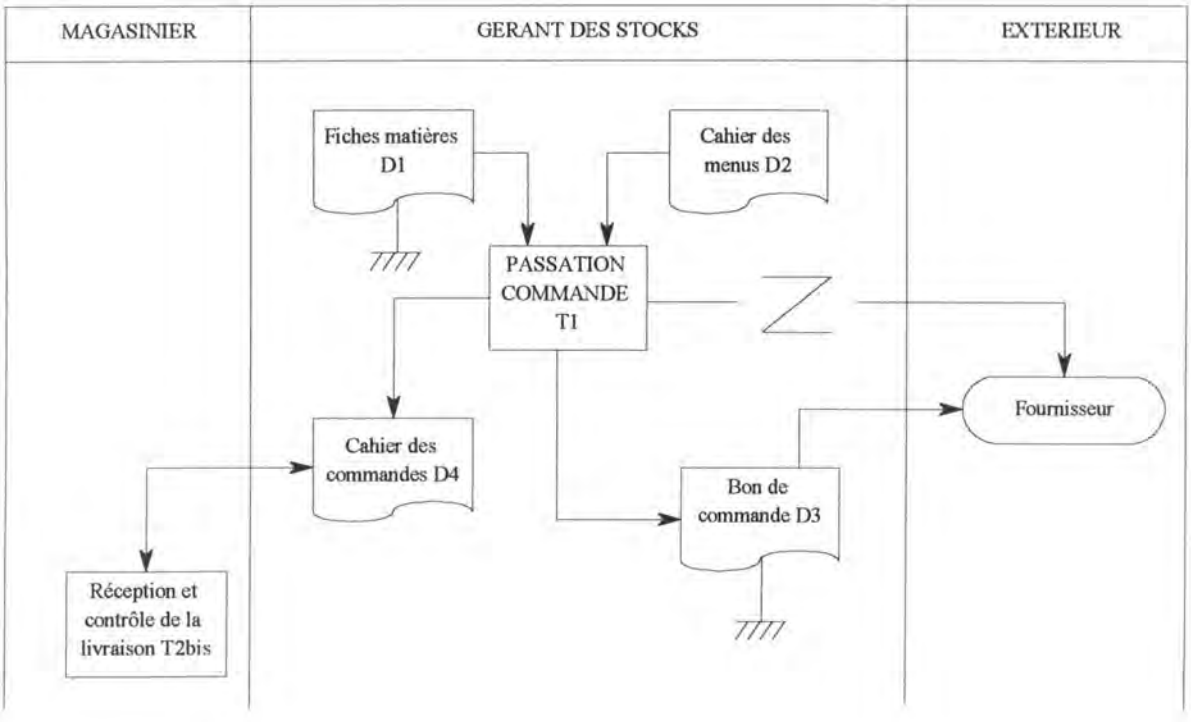
Une autre représentation peut être envisagée par la création d'une *PU-Collection*. Dans ce cas, les différentes *processing units* pouvant faire l'objet d'un traitement parallèle, sont regroupées en une collection.

4. Validation de la proposition

4.1 Le modèle Merise

- ***Etude de l'existant***

L'exemple repris ci-dessous est l'Exemple 2-1 décrit dans le chapitre 2.



La première étape consiste à créer une *processing unit* par traitement présent dans la représentation. Nous relions ces *processing units* avec leurs paramètres, en indiquant s'il s'agit d'un paramètre en entrée ou/et en sortie.

Dans le cas du traitement "passation commande", nous avons deux paramètres en entrée et deux en sortie ; tandis que le traitement "réception et contrôle de la livraison" n'a qu'un seul paramètre qui est à la fois en entrée et en sortie.

Un autre élément doit encore être ajouté à la représentation. Il s'agit de la notion d'acteur. Il nous faut en effet préciser qui agit sur une tâche ou un document.

Cela se représente au moyen de l'entité *actor*. Nous avons donc deux acteurs qui agissent sur ce premier traitement : le gérant des stocks et le fournisseur. Nous créons une première entité

Actor représentant le gérant des stocks. Nous la relions ensuite à la *processing unit* "Passation commande" ainsi qu'à ses paramètres (voir Figure 4-1).

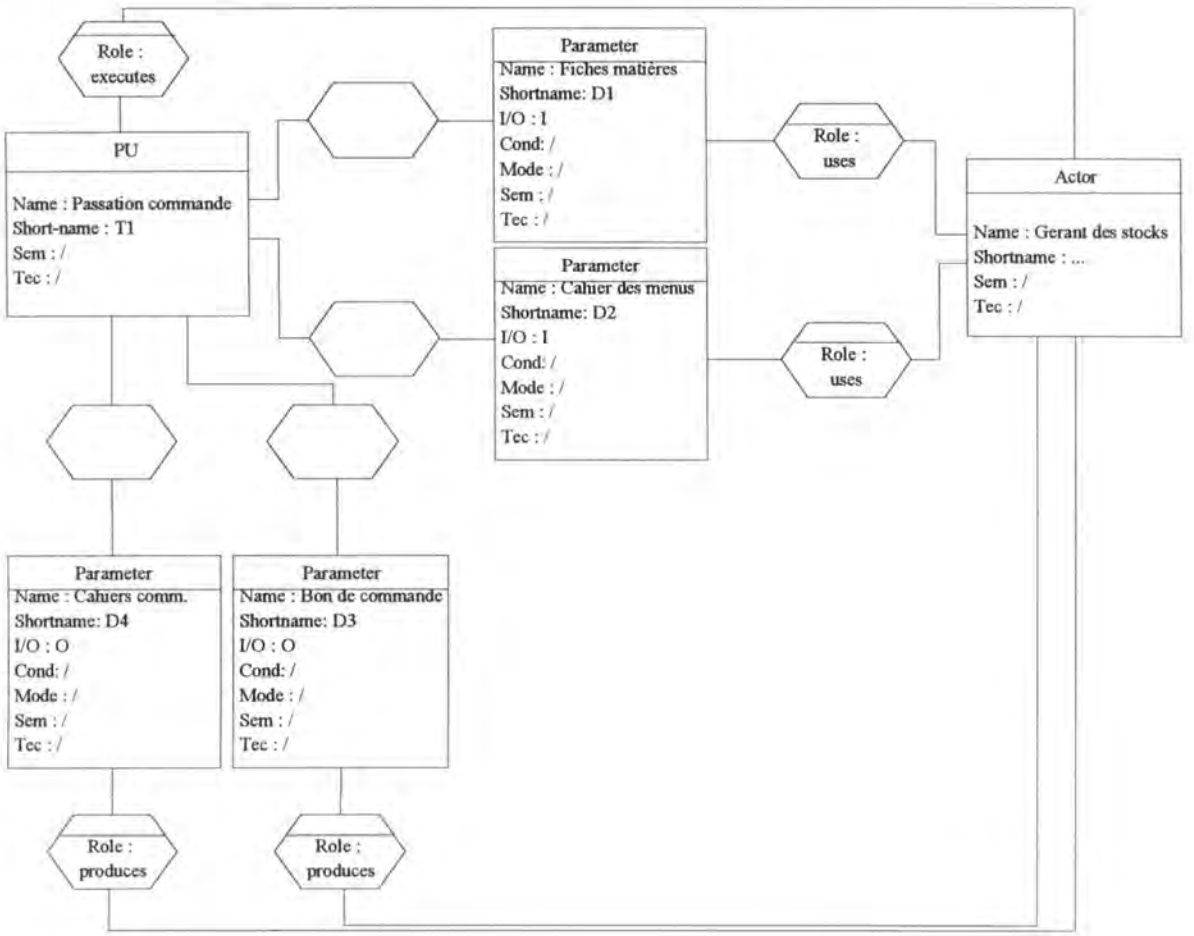


Figure 4-1 : Merise - Lien entre la partie traitement et la partie donnée

Le lien entre la *processing unit* "Passation commande" et le fournisseur est un peu plus compliquée. En effet, nous n'avons pas la possibilité d'indiquer qu'une *processing unit* envoit un message vers un acteur. Pour contrer cette difficulté, nous créons une *processing unit* "Transm_1", qui est reliée à "Passation commande" via la relation *PU-Relationship* (voir Figure 4-2).

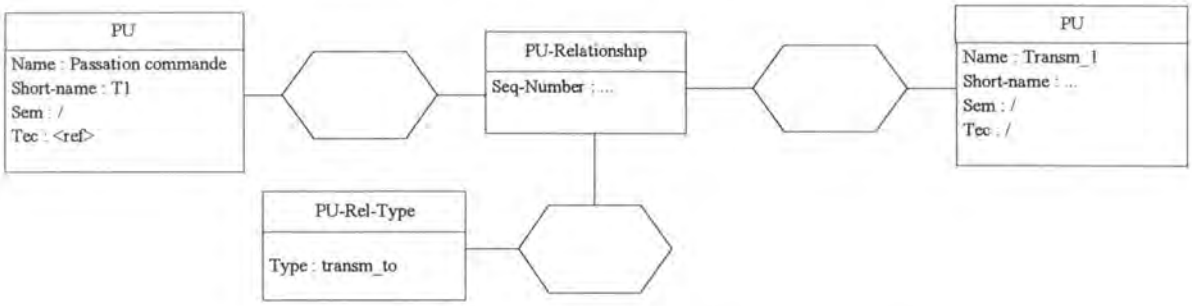


Figure 4-2 : Merise - Transmission d'un document vers un fournisseur

Nous relions ensuite cette nouvelle *processing unit* avec l'entité *Actor* "Fournisseur" (voir Figure 4-3).

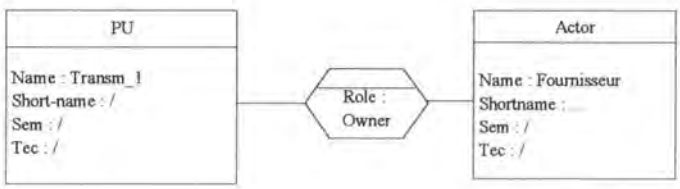


Figure 4-3 : Merise - Transmission d'un document vers un fournisseur

Le second traitement se représente de manière similaire (voir Figure 4-4).

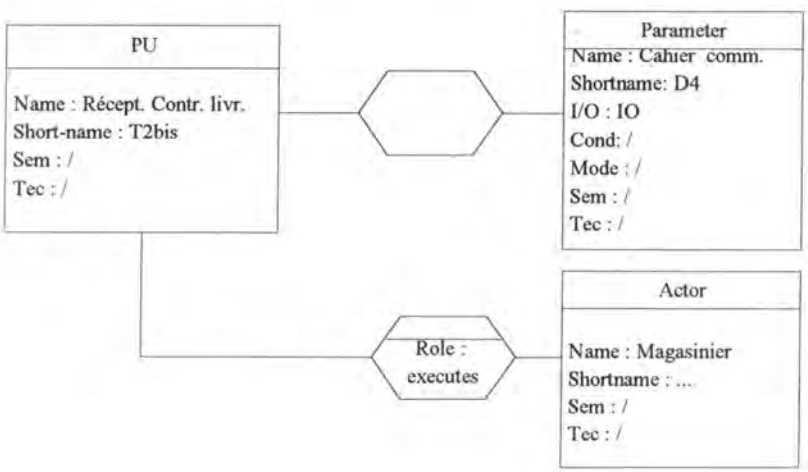


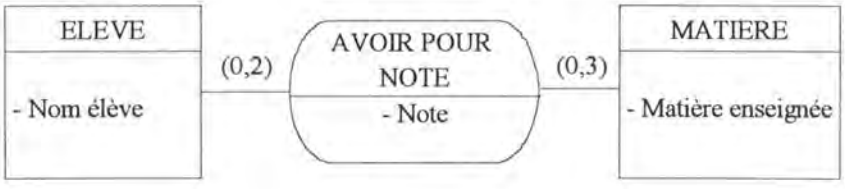
Figure 4-4 : Lien entre la partie traitement et la partie donnée

Un seul point de l'exemple n'a pas encore été représenté. Il s'agit de la possibilité d'indiquer qu'un document peut être classé. Cela ne va pas poser de problème : en effet, le document est

représenté par une *Entity-type*. Il nous suffit alors de considérer que le classement est une caractéristique du document, et donc de stocker cette information dans la partie *Sem* de l'entité.

- ***Modèle conceptuel des données***

Nous reprenons ici l'Exemple 2-2.



Analysons maintenant la représentation utilisée dans le modèle conceptuel des données.
Il est assez aisé de le représenter dans le repository, sans même faire appel aux structures que nous avons ajoutées. En effet, il nous suffit de créer une relation entre deux *Entity-type*s : "élève" et "matière" (voir Figure 4-5).

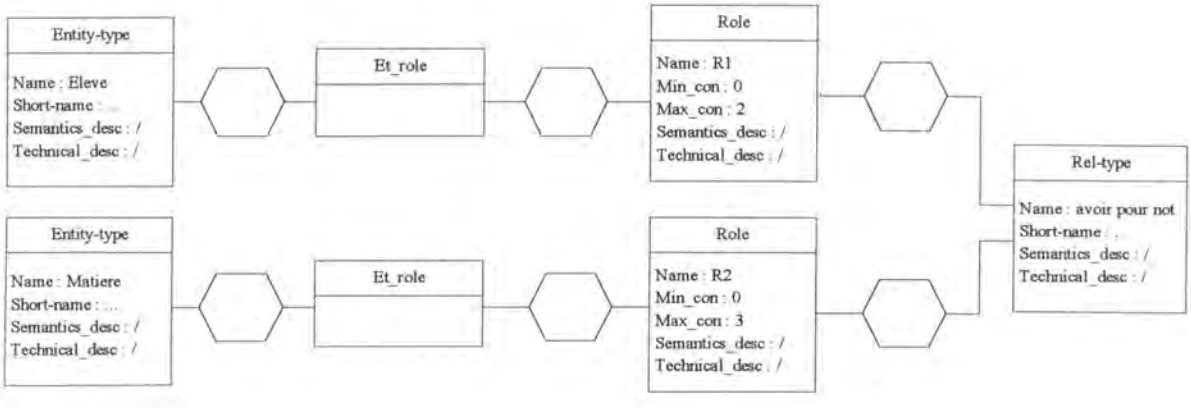


Figure 4-5 : Merise - Relation entre deux entités

Il nous faut encore relier ces *Entity-type* avec leur attribut. Pour cela, il nous suffit d'utiliser la relation *Owner-Att* (voir Figure 4-6).

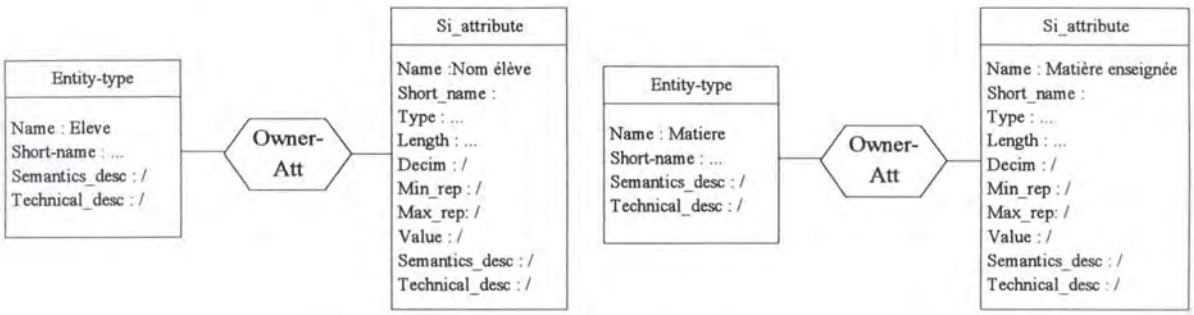
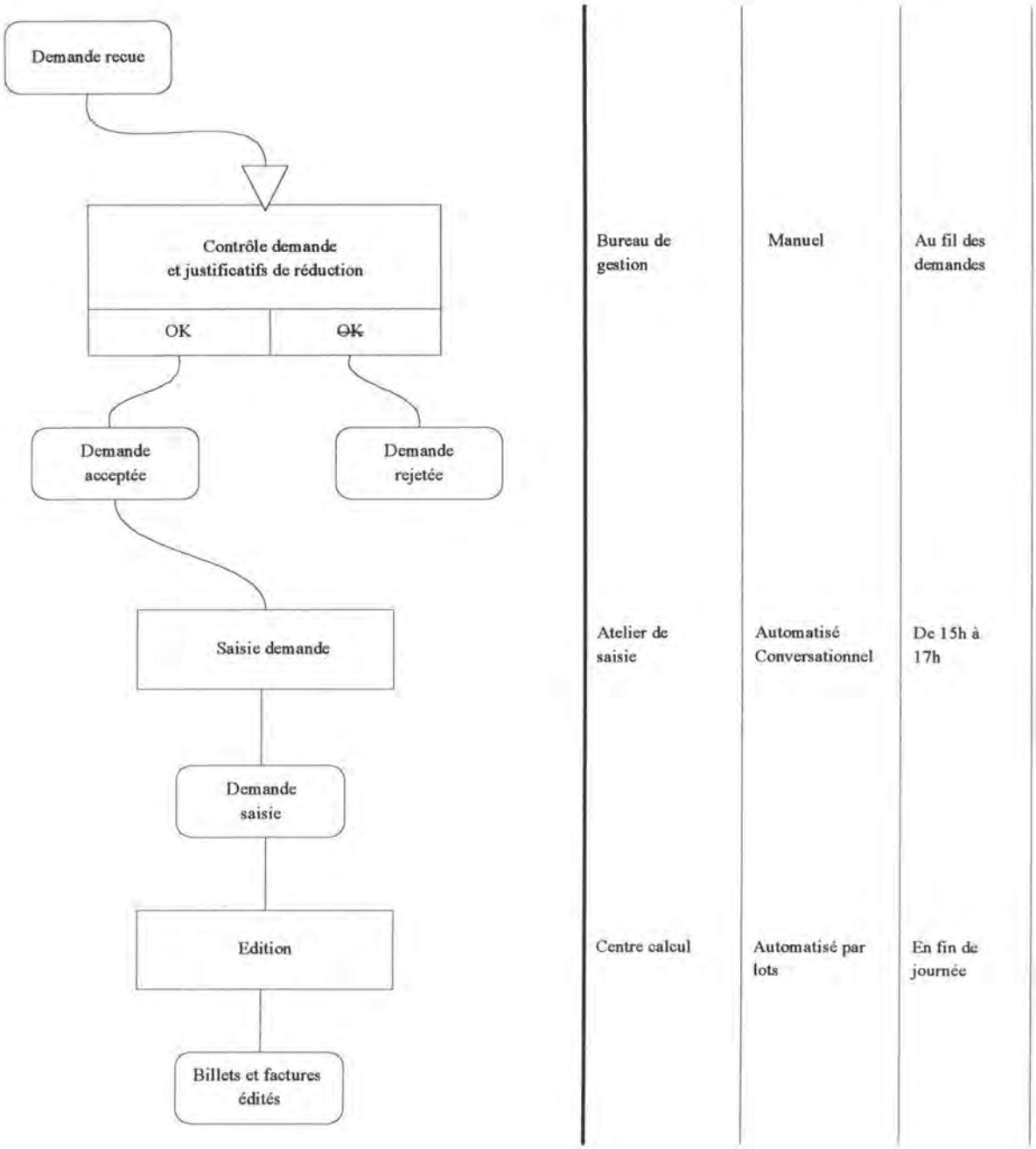


Figure 4-6 : Merise - Décomposition d'une entité en attribut

- ***Modèle organisationnel des traitements***

L'exemple qui va être utilisé ici est l'Exemple 2-3.



Le dernier mode de représentation que nous avons montré se situe au niveau organisationnel des traitements. Pour le représenter, nous créons tout d'abord une *processing* unit de type *program* (voir Figure 4-7).

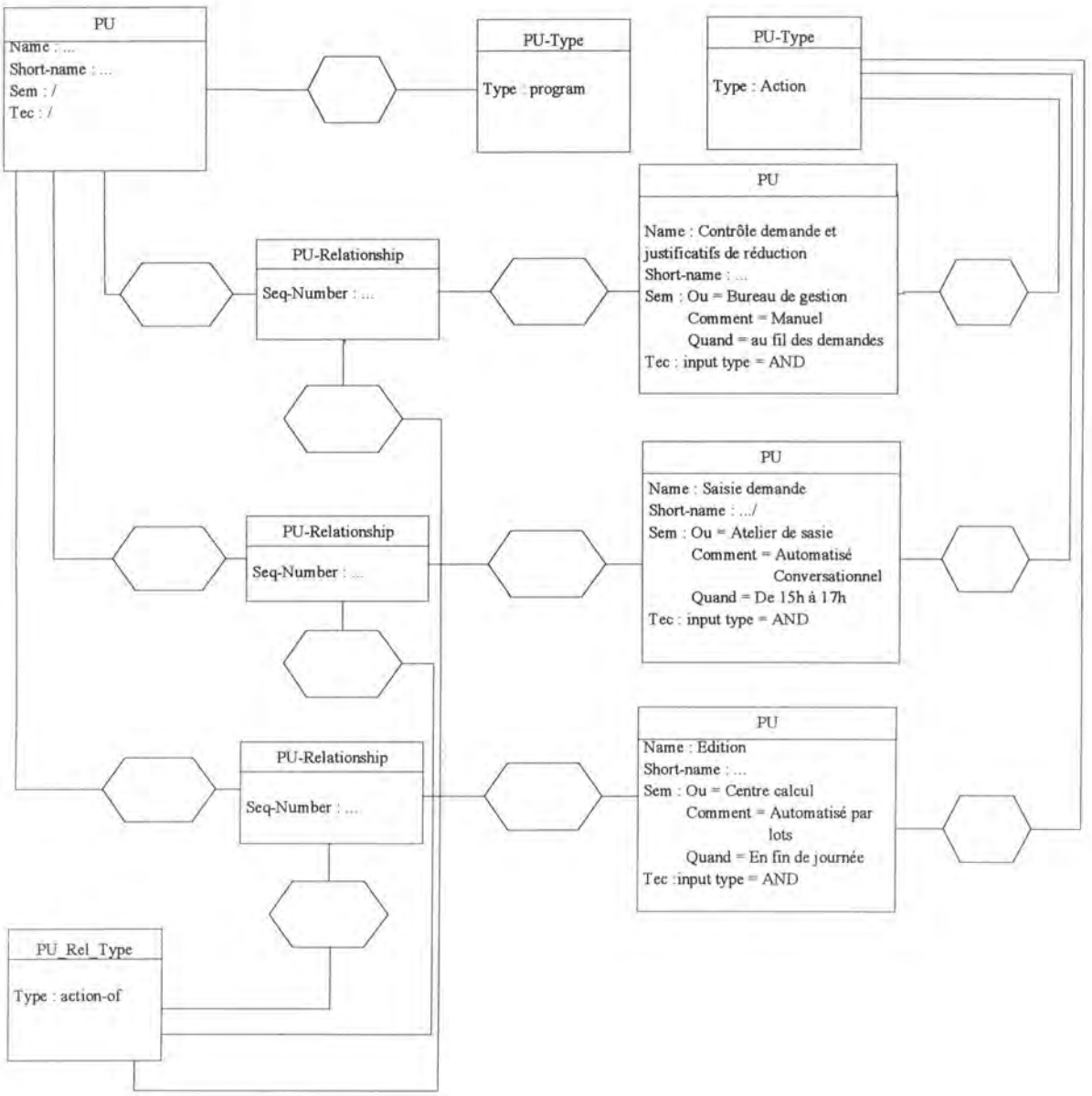


Figure 4-7 : Décomposition d'un programme en actions

Cette *processing unit* est alors décomposée en sous-*processing unit* représentant chacune un traitement. Comme précédemment, nous relions ces sous-*processing units* avec leurs paramètres.

Il reste encore à représenter les informations relatives au poste de travail, à la nature du traitement et à sa chronologie de déclenchement (voir Figure 4-8).

Nous stockons cette information dans la partie sémantique de la *processing unit*.

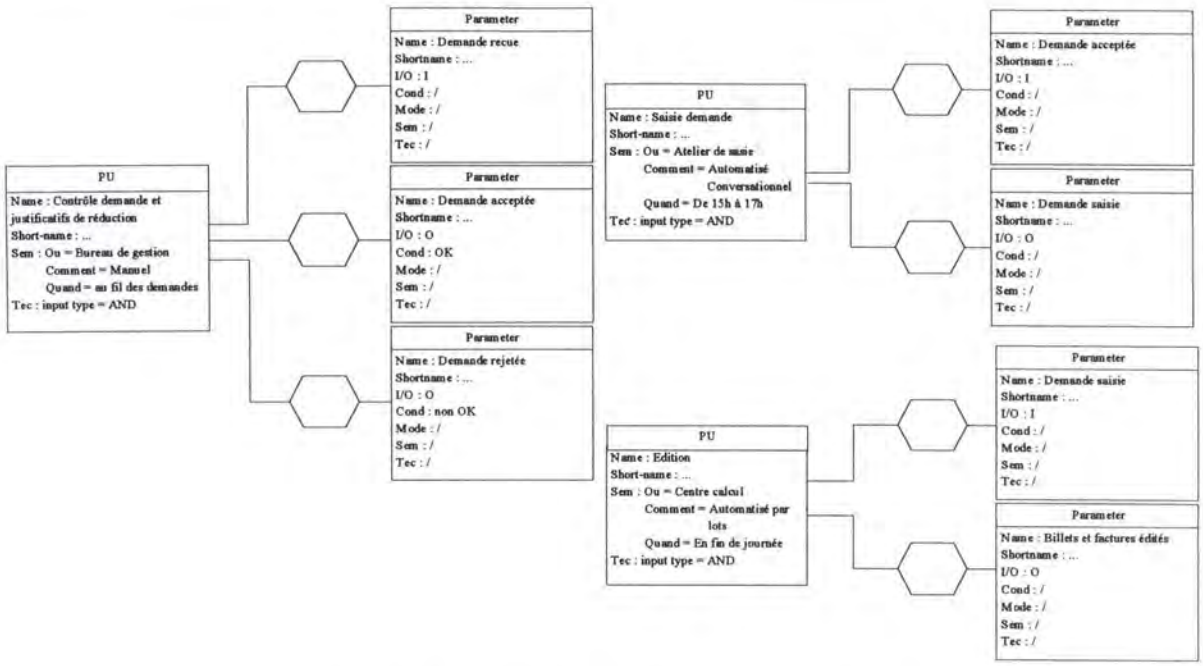
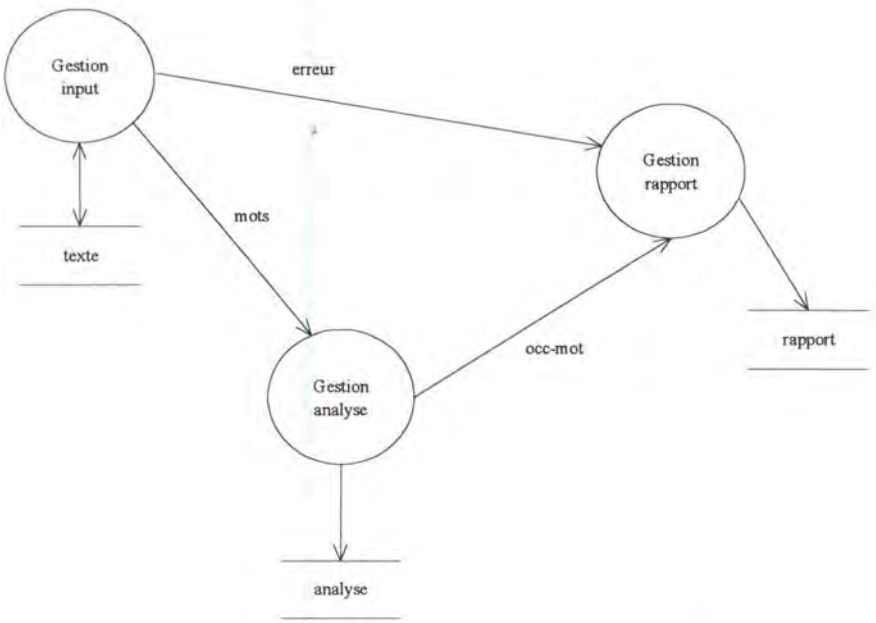


Figure 4-8 : Merise - Lien entre la partie traitement et la partie donnée

4.2 *Dataflow Diagram*

Analyse Texte



Pour représenter le programme "Analyse Texte" (voir Exemple 2-4), dont le schéma est repris ci-dessus, la première opération à effectuer est de créer une *processing unit* contenant l'ensemble de la spécification. Cette *processing unit* est de type "program".

Nous créons ensuite une *processing unit* par fonction présente dans la spécification, et nous la relions à la *processing unit* "Analyse-texte".

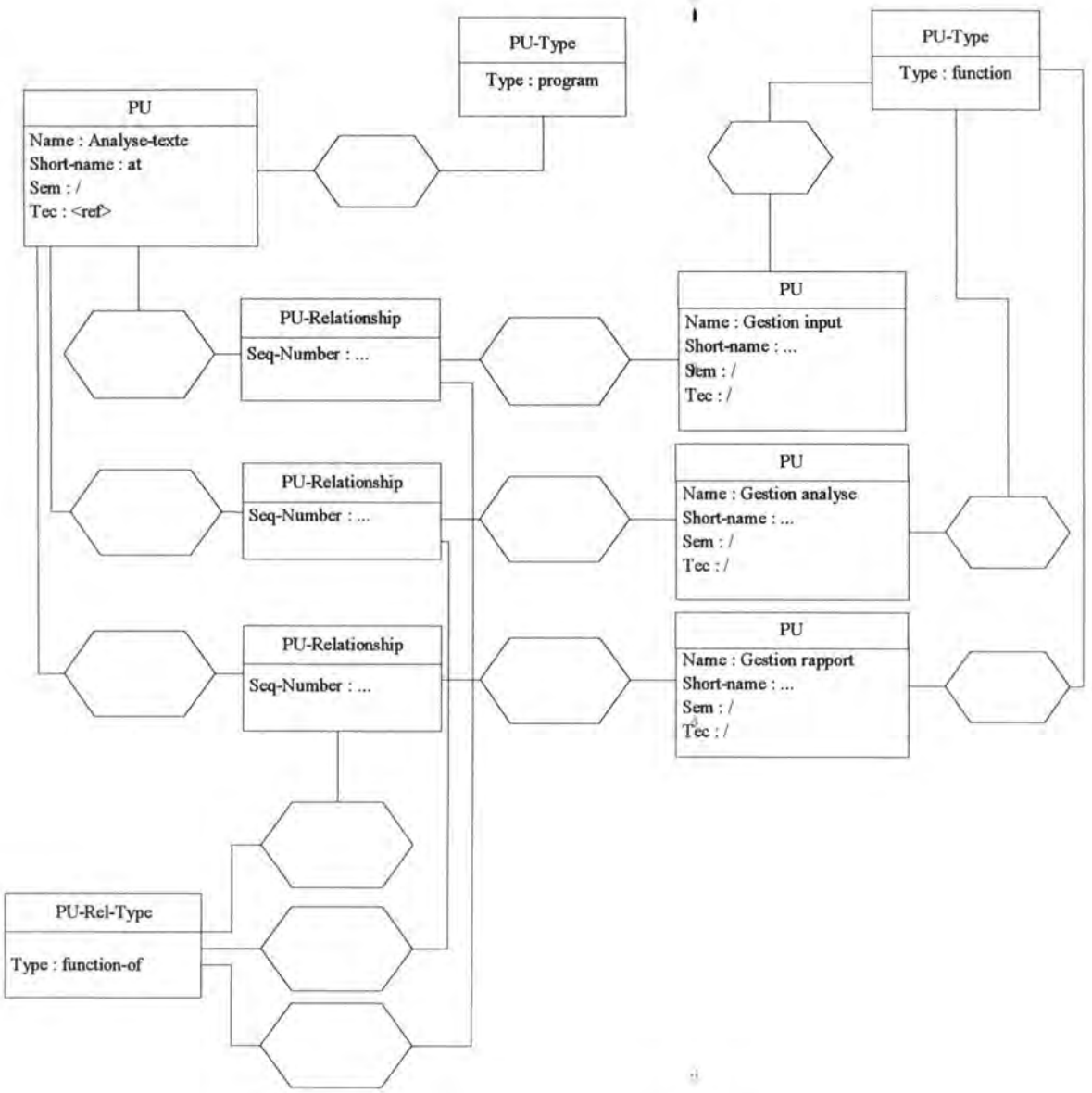


Figure 4-9 : DFD - Décomposition d'un programme en fonctions

Il nous faut maintenant relier les différentes *processing units* avec leurs données en entrée et en sortie. Nous réalisons cela au moyen d'entités *parameter*. Ces entités sont alors reliées à leur *Entity-type* correspondant (représenté par l'entité *Generic-Object*).

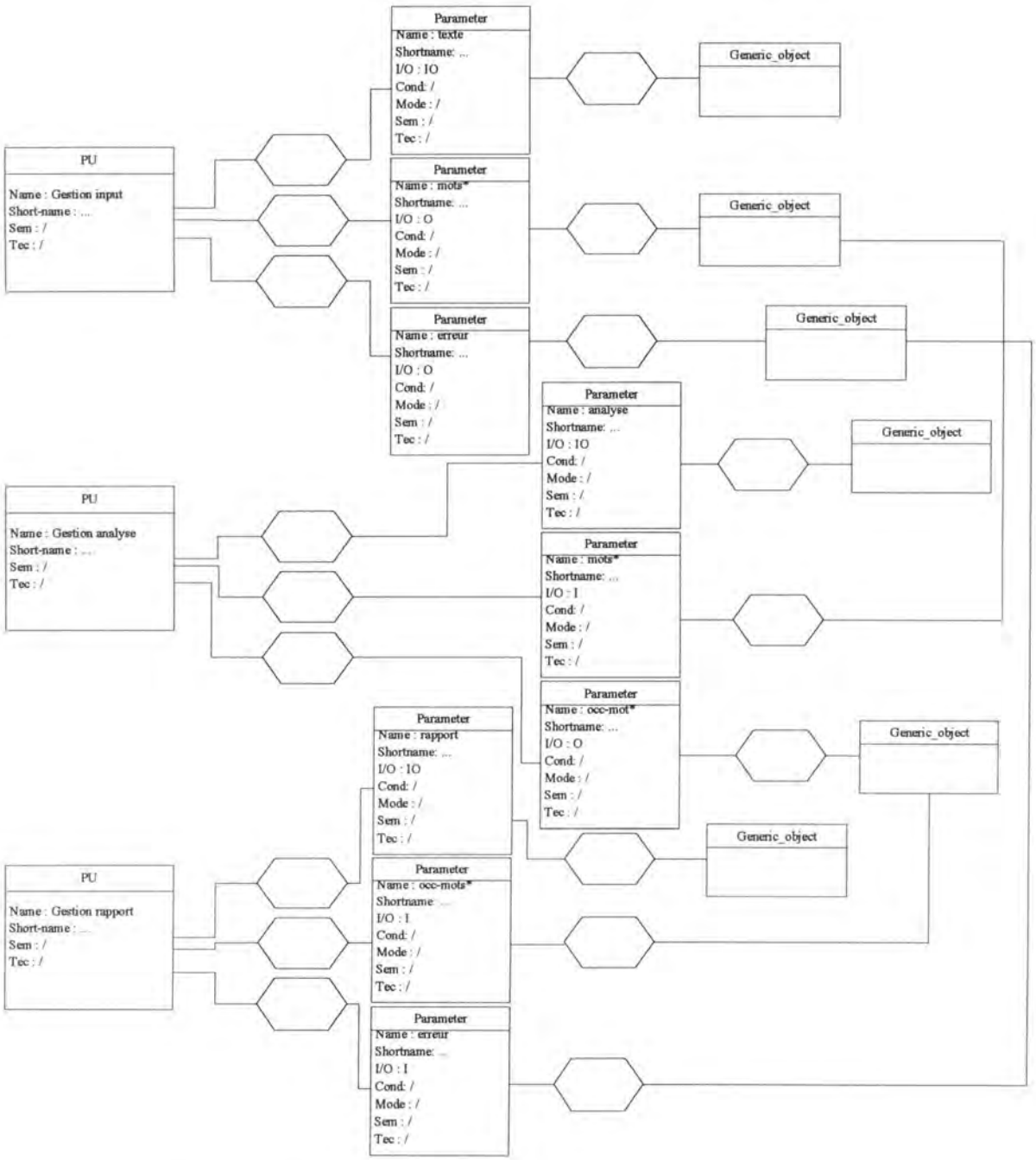
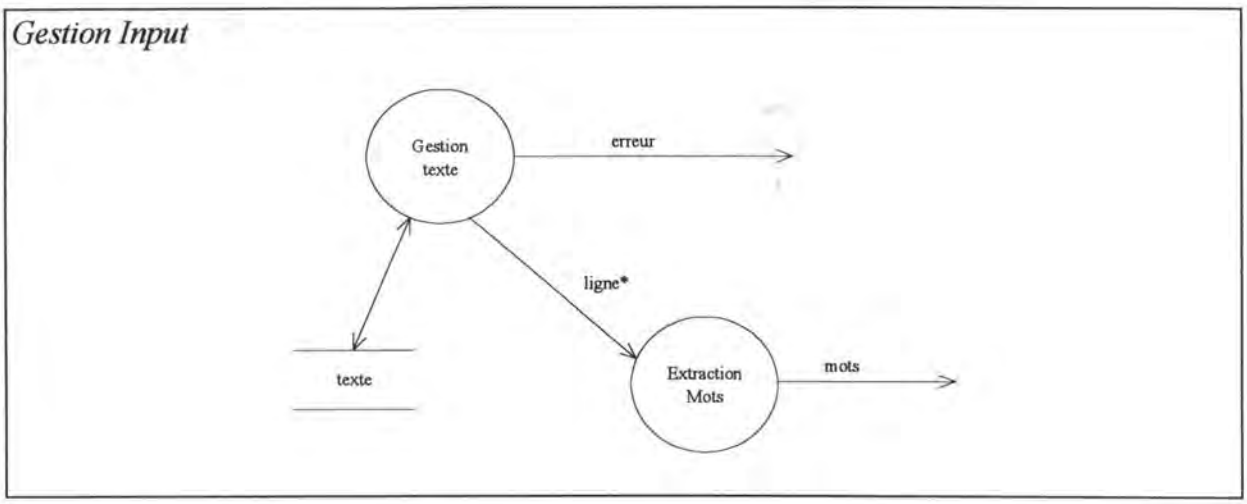


Figure 4-10 : DFD - Lien entre la partie traitement et la partie donnée

La spécification du programme de l'Exemple 2-4 est maintenant représentée. Mais, comme nous l'avons déjà dit, les fonctions qui composent cette spécification peuvent être raffinées (voir Exemple 2-5).



Il nous faut donc créer un lien entre la fonction présente dans le premier schéma et sa décomposition. Cette relation est illustrée à la Figure 4-11.

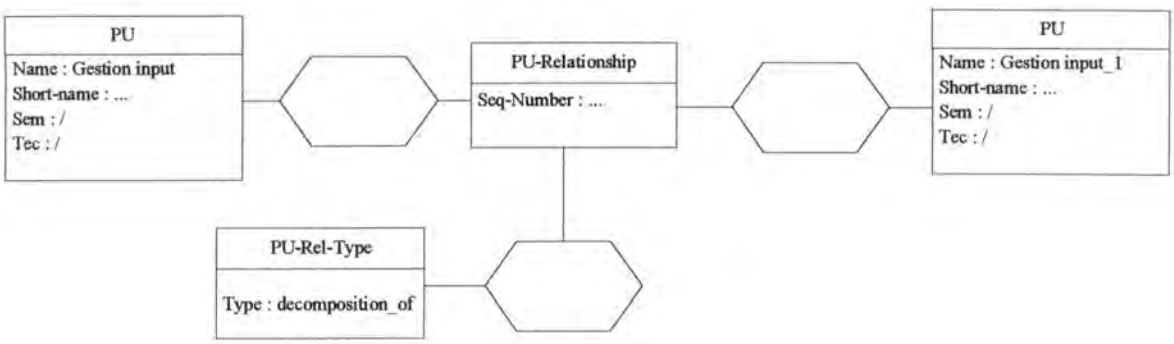


Figure 4-11 : DFD - Raffinement d'une fonction

La fonction "Gestion input_1" se représente de la même manière que "Gestion input".

Reprenons l'exemple du chapitre 2 (voir Exemple 2-6) concernant la spécification des fonctions en pseudo-code :

```
(out, texte') = Gestion-texte(texte)

        LIGNE = SEQ[CHAR]
        texte, texte', l, t, t', t'' : SEQ[LIGNE]

        PRE
```

```
POST
        out = 'Fichier vide' <= Eof(texte)
        (out*, texte*) = Extraction(texte) <= not eof(texte)

        (t', l*) = Extraction(t)
                <=> IF Eof(t)
                        THEN l* = [] and t'=t
                        ELSE (t'', l) = Read(t)
                                l* = Append(l, Extraction(t''))
```

La spécification des fonctions de plus bas niveau (voir exemple ci-dessus) en terme de pré et post-conditions est stockée dans l'attribut *Tec* de la *processing unit* correspondant à la fonction.

Nous ne représentons pas toutes les fonctions, ni toutes les décompositions, dans la mesure où le procédé utilisé ne change pas.

4.3 *Types abstraits algébriques*

Pour cette partie, nous reprenons l'Exemple 2-7 décrit dans la partie relative à la présentation du langage.

PileVide :		→ Pile-Mots
Empiler :	Pile-Mots	→ Pilè-Mots
Dépiler :	Pile-Mots	→ Pile-Mots
Tête :	Pile-Mots	→ Mots
Pvide? :	Pile-Mots	→ Bool

Tête (Empiler (pile, m)) = m
Pvide? (Empiler (pile, m)) = Faux
Pvide? (PileVide ()) = Vrai
Dépiler (Empiler (pile, m)) = pile
Tête (PileVide ()) = erreur
Pré : Tête (p) : ¬ Pvide? (p)
Dépiler (PileVide ()) = PileVide
Pré : Dépiler (p) : ¬ Pvide? (p)

Les définitions des objets seront représentées par des *entity-types*.

Comme dans les modèles vus précédemment, nous allons créer une *processing unit* représentant l'entièreté de la spécification. Par la suite, une *processing* unit sera créée pour chaque axiome.

Les paramètres indécomposables de l'axiome seront représentés par des entités *parameter* tandis que les expressions décomposables seront représentées par une *processing unit* reliée à la précédente par une relation de type « axiom-of ».

C'est ainsi que pour représenter le premier axiome de l'exemple ci-dessus nous obtenons le schéma suivant (voir Figure 4-12) :

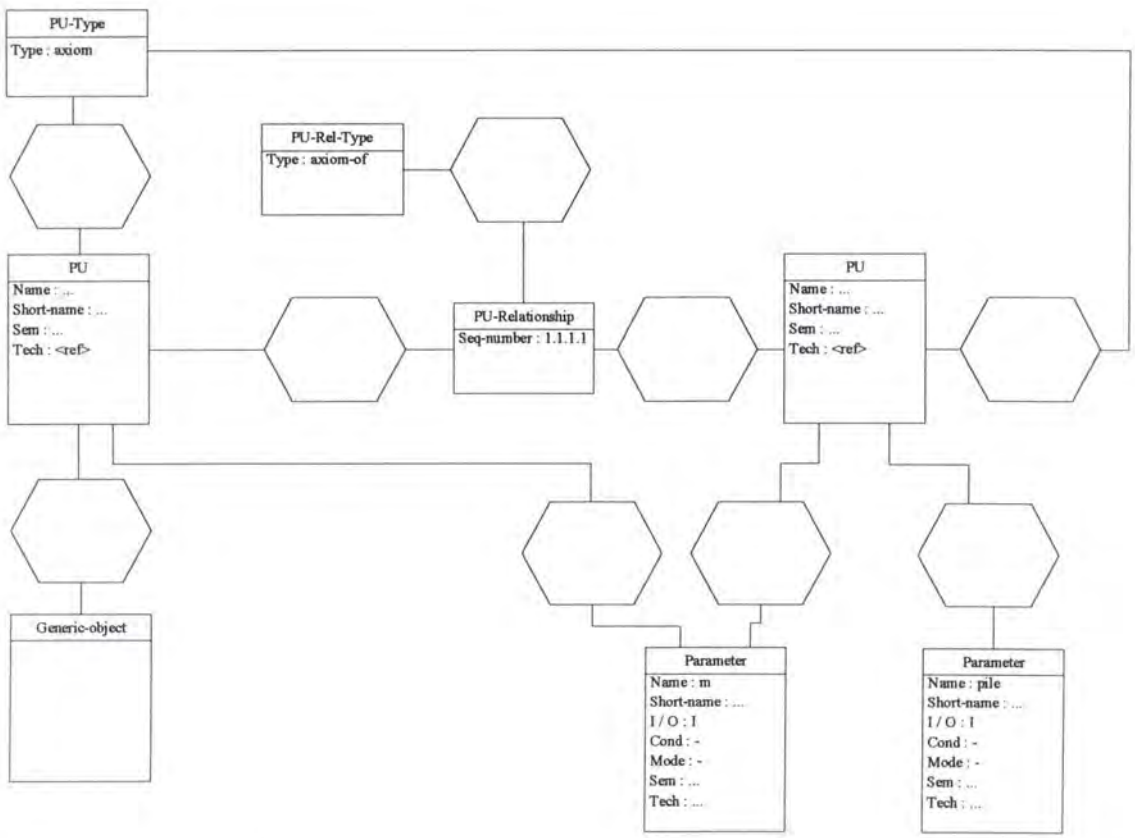


Figure 4-12 : TAA - Décomposition d'un axiome

Le deuxième axiome se représente de façon similaire (voir Figure 4-13).

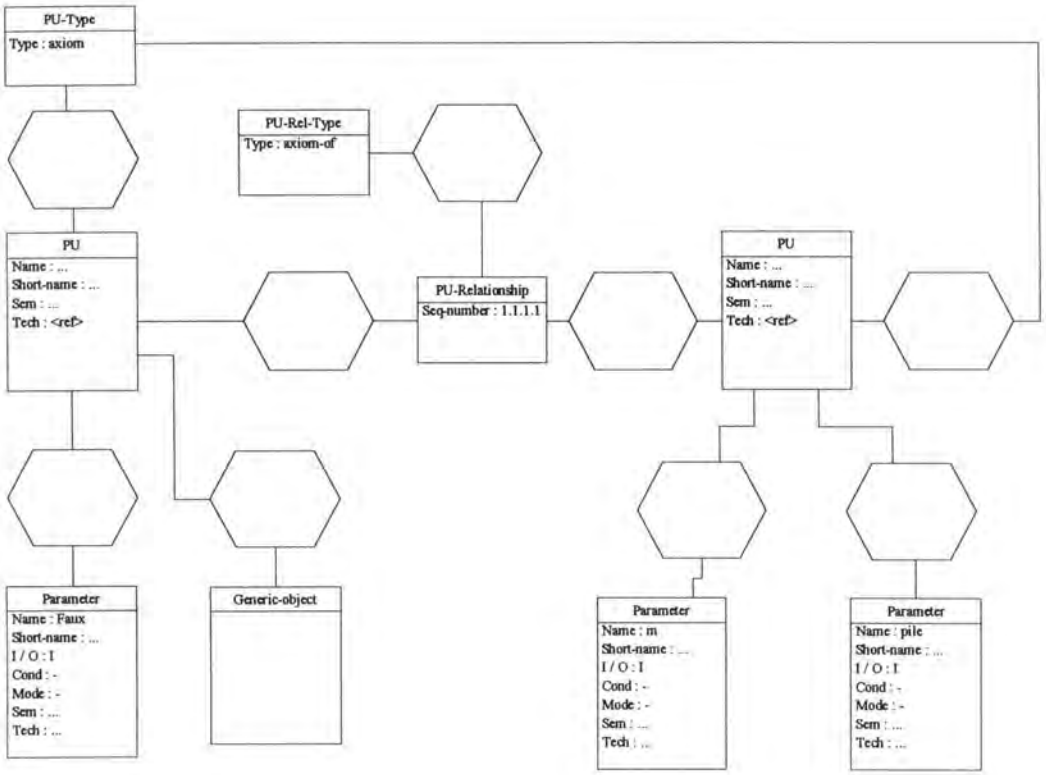


Figure 4-13 : TAA - Décomposition d'un axiome

Il nous reste maintenant à représenter les deux mécanismes de structuration, à savoir la généricité et l'héritage.

a) La généricité

L'exemple, repris ci-dessous afin d'illustrer le mécanisme de généricité, est l'Exemple 2-9 du chapitre 2.

F-Vide :		$\rightarrow$ FILE
Arrivée :	FILE $\times$ X	$\rightarrow$ FILE
Départ :	FILE	$\rightarrow$ FILE
A-Servir :	FILE	$\rightarrow$ X

Afin d'illustrer l'instanciation d'un type générique, reprenons l'Exemple 2-10 :

FILE-CLIENT = FILE [CLIENT]
FILE-FOURNISSEUR = FILE [FOURNISSEUR]

L'objet construit est représenté comme précédemment par une *entity-type*. Le *generic-object* correspondant est relié à une *processing unit* représentant l'ensemble des axiomes. Cette *processing unit* est reliée à d'autres *processing units*, chacune représentant un axiome. Cet axiome est alors, si nécessaire, décomposé en axiomes plus simples. Lors de l'instantiation, l'*entity-type* représentant l'objet générique sera reliée par un *rel-type* à l'*entity-type* modélisant l'instance (voir Figure 4-14). La structure de l'instance est la même que celle de son type générique.

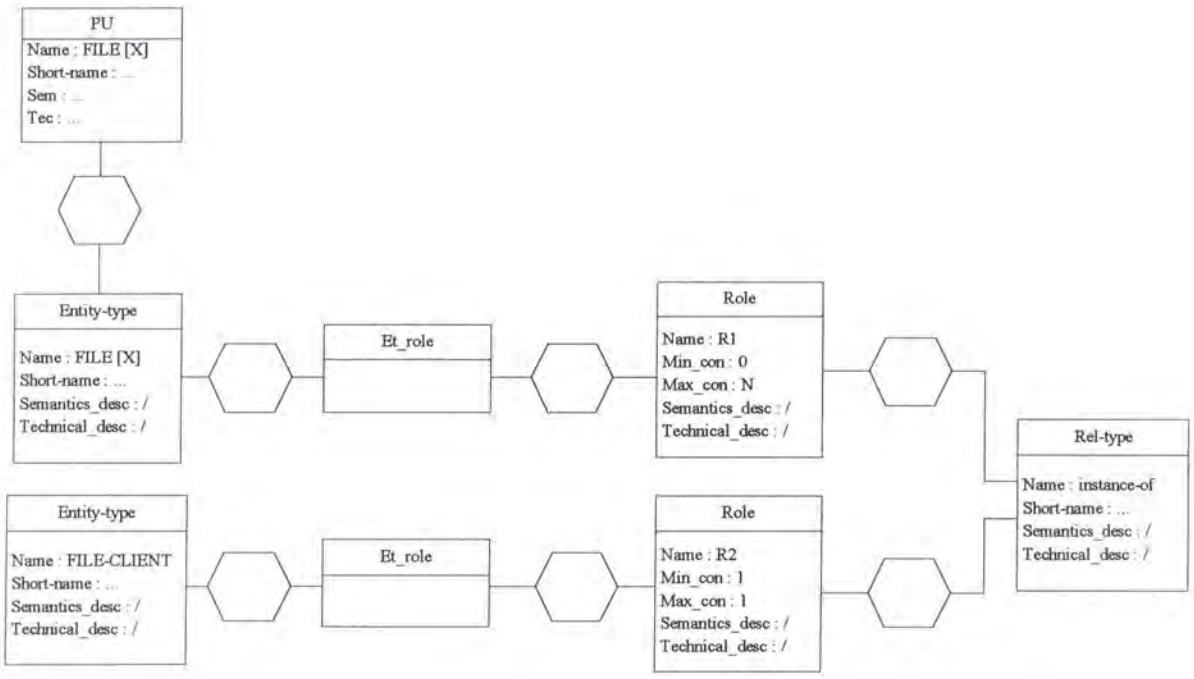


Figure 4-14 : TAA - Instanciation d'un type générique

b) L'héritage

Afin d'illustrer ce mécanisme, reprenons l'Exemple 2-11 :

```
RECTANGLE is CP  (Point1 : POINT,
                  Point2 : POINT,
                  Point3 : POINT,
                  Point4 : POINT)
{CP représente un produit cartésien. On suppose que le type POINT a été défini précédemment.}

Périmètre (r) = p
        r : RECTANGLE
        p : INTEGER

        Pré :
        Post :p = 2 * (côté1 + côté2)
              côté1 = Ord (Point2 (r)) - Ord (Point1 (r))
              côté2 = Abs (Point3 (r)) - Abs (Point2 (r))

CARRE isa RECTANGLE
Invariant : Ord (Point2 (c)) - Ord (Point1 (c)) = Abs (Point3 (c)) - Abs (Point2 (c))

Périmètre (c) = p
        c : CARRE
        p : INTEGER

        Pré :
        Post :p = 4 * côté
              côté = Ord (Point2 (c)) - Ord (Point1 (c))
```

Pour modéliser l'héritage, nous utiliserons le concept de sous-type de DB-Main (voir Figure 4-15).

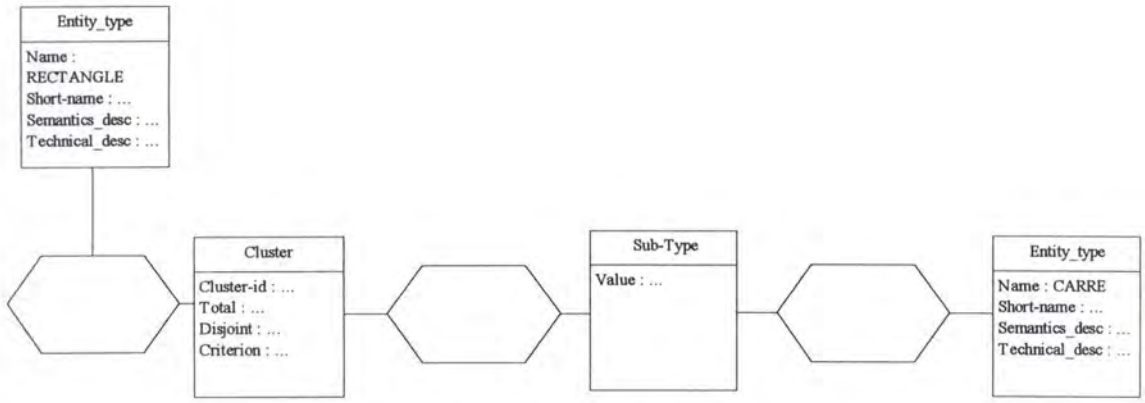


Figure 4-15 : TAA - Mécanisme d'héritage

4.4 Telos

Nous nous basons ici sur l'Exemple 2-12.

```
TELL CLASS Paper IN SimpleClass WITH
        attribute
                author : String;
                referee : String;
                title : String;
                pages : 1..100
END
```

Pour représenter la définition de la classe "Paper", nous créons une Entity-Type de nom *Paper*. Nous utilisons l'attribut *Tec* pour indiquer qu'il s'agit d'une classe. Cette *Entity-Type* est alors décomposée en ses différents attributs (voir Figure 4-16).

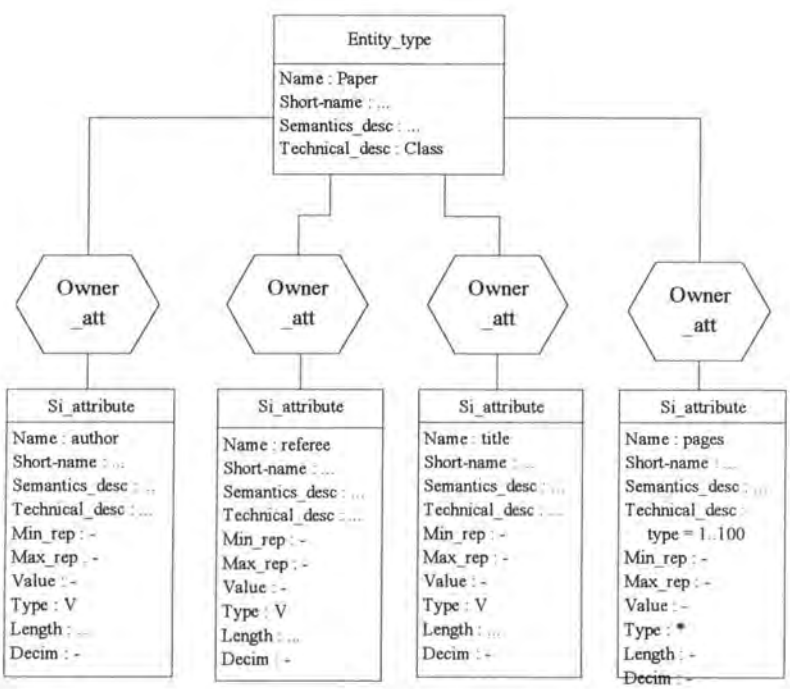


Figure 4-16 : Telos - Représentation d'une classe

Nous devons maintenant représenter le fait que la classe "Paper" est une instance de la classe "SimpleClass". Pour cela, nous créons une entité *Rel-type*. Cette entité est alors reliée d'une part à la classe "Paper" et d'autre part à la classe "SimpleClass". Le type de relation entre ces deux classes est stocké dans l'attribut *sem* de l'entité *Role* (voir Figure 4-17).

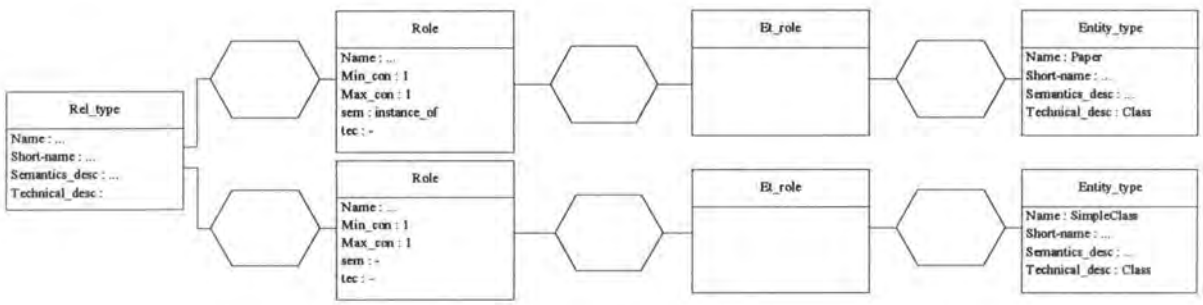


Figure 4-17 : Telos - Représentation d'une classe

La suite de cet exposé repose sur l'Exemple 2-13.

```
TELL TOKEN martian IN Paper WITH
        author
                firstauthor     : Stanley;
                                : LaSalle;
                                : Wong
        title
                : 'The MARTIAN system'
END
```

La définition d'un "*token*" se fait d'une manière similaire. Tout d'abord, nous créons une *entity-type* de nom "martian", que nous décomposons en ses différents attributs, tout en indiquant qu'il s'agit d'un "*token*" (voir Figure 4-18).

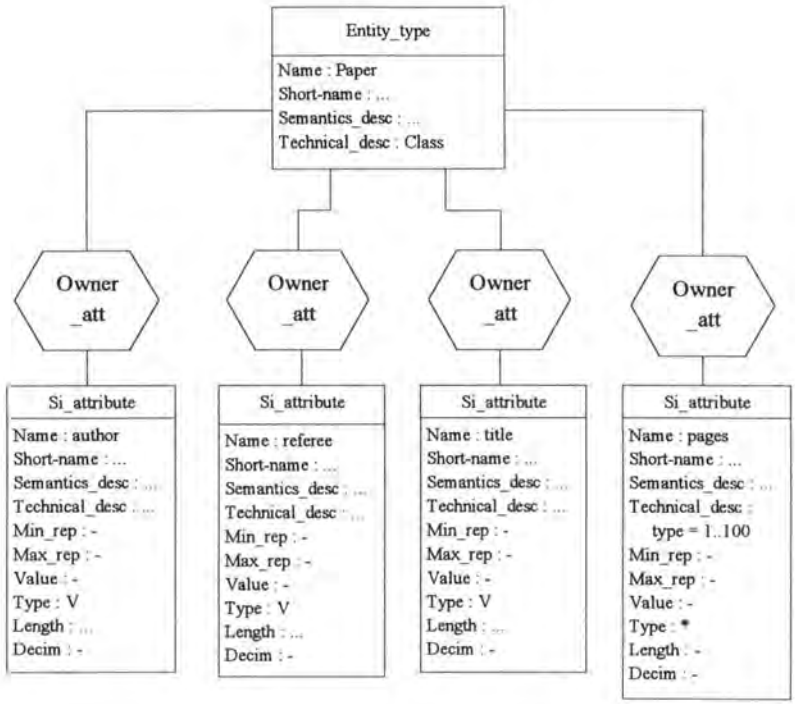


Figure 4-18 : Telos - Représentation d'un token

Nous créons ensuite une entité *Rel-type* pour indiquer que *martian* est une instance de la classe *Paper* (voir Figure -194-20).

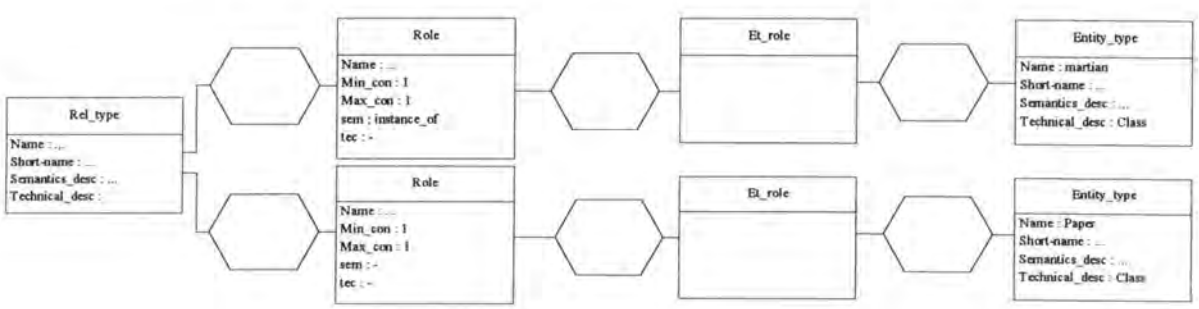


Figure -194-20 : Telos - Représentation d'un token

Un procédé similaire est utilisé pour représenter la notion de spécialisation. Pour analyser ce mécanisme, reprenons l' Exemple 2-14.

```
TELL CLASS AcceptedPaper IN SimpleClass ISA Paper WITH
        attribute
                pages : 1..15
                session : Integer
END
```

Nous décomposons tout d'abord l'*Entity-type* "AcceptedPaper" en ses différents paramètres (voir Figure 4-21).

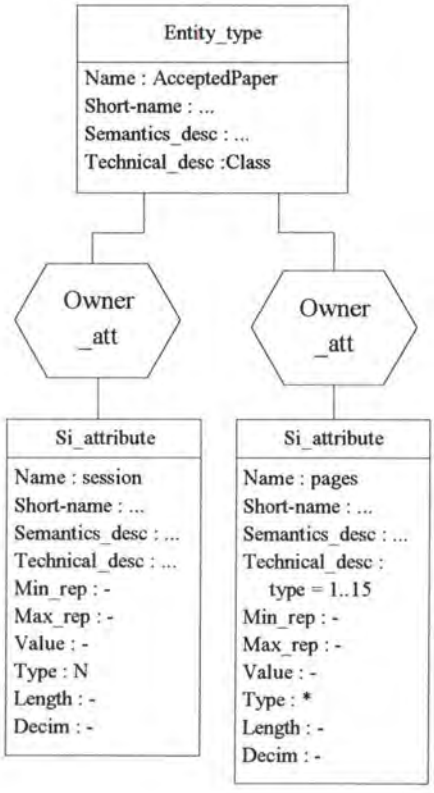


Figure 4-21 : Telos - Spécialisation d'une classe

Nous créons, comme précédemment, une entité *Rel-type* pour représenter le fait que *AcceptedPaper* est une instance de la classe *SimpleClass* (voir Figure 4-22).

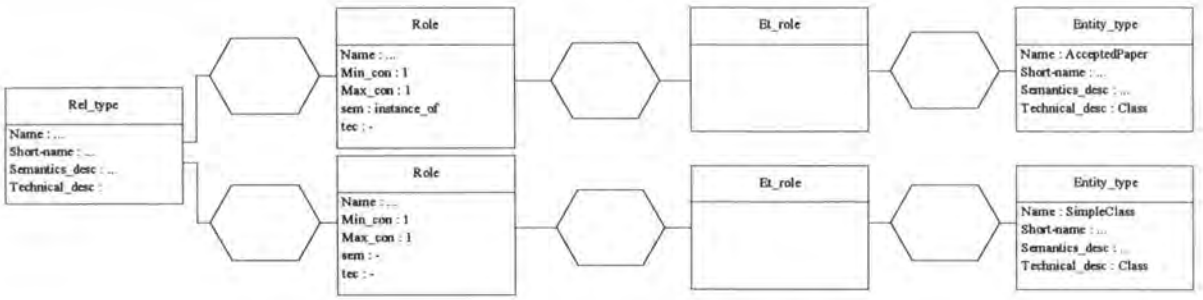


Figure 4-22 : Telos - Spécialisation d'une classe

Mais nous devons encore indiquer que la classe *AcceptedPaper* est une spécialisation de la classe *Paper*. Pour cela nous utilisons une structure déjà présente dans le repository, à savoir la notion de *cluster* (voir Figure 4-23).

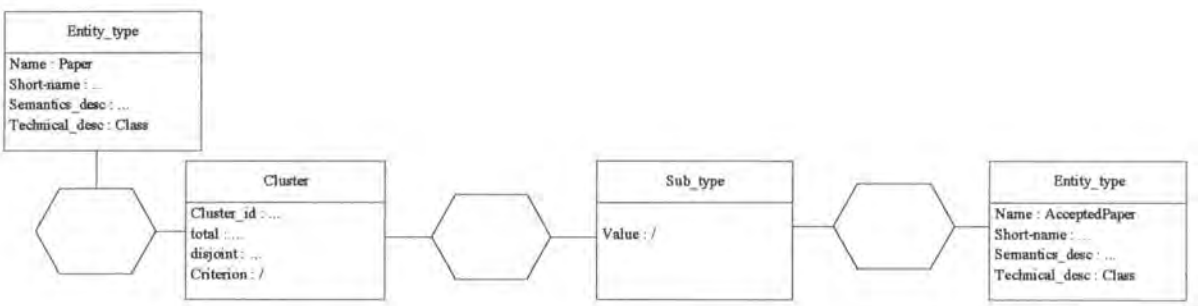


Figure 4-23 : Telos - Spécialisation d'une classe

L'ajout de contraintes d'intégrité ou de règles de déduction à une classe (voir Exemple 2-15 ci-dessous) ne pose pas de problème.

```
TELL CLASS Paper IN SimpleClass WITH
        integrityContraint
                :$      (∀ y/Person)
                        (y ∈ this.author ⇒ ¬y∈ this.referee) $
        deductiveRule
                :$      (∀ x/Paper)(∀ z/Address)
                        (z ∈ x.author.address ⇒ z ∈ x.replyAddress) $
END
```

Il suffit de relier cette entité (représentée par l'entité *Generic-Object* dans la Figure 4-24) à deux *processing units* représentant respectivement la contrainte d'intégrité et la règle de déduction. Le contenu (ou une référence au contenu) de la contrainte et de la règle est stocké dans l'attribut *Tec* de la *processing unit*.

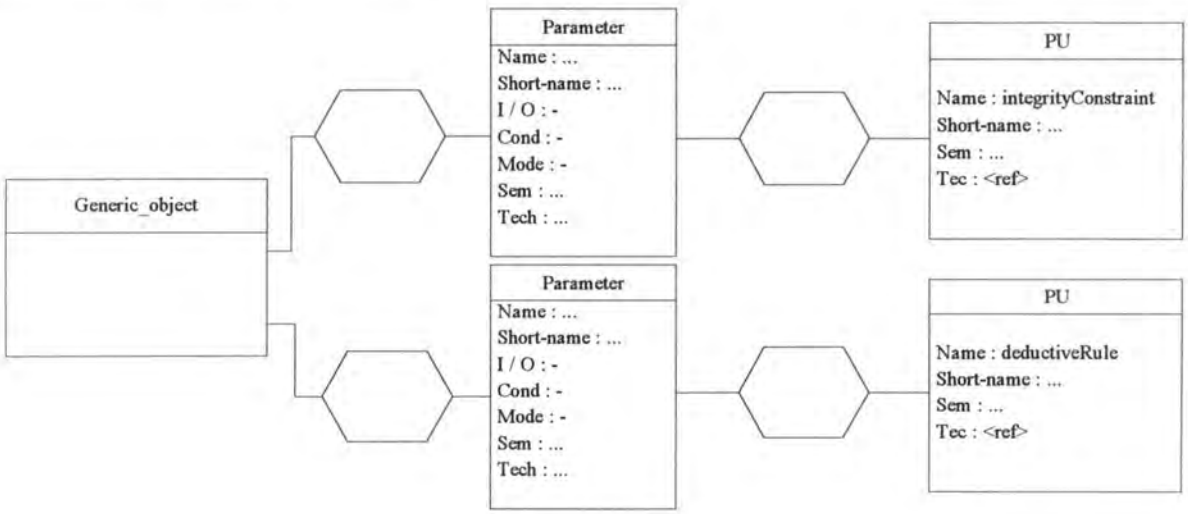
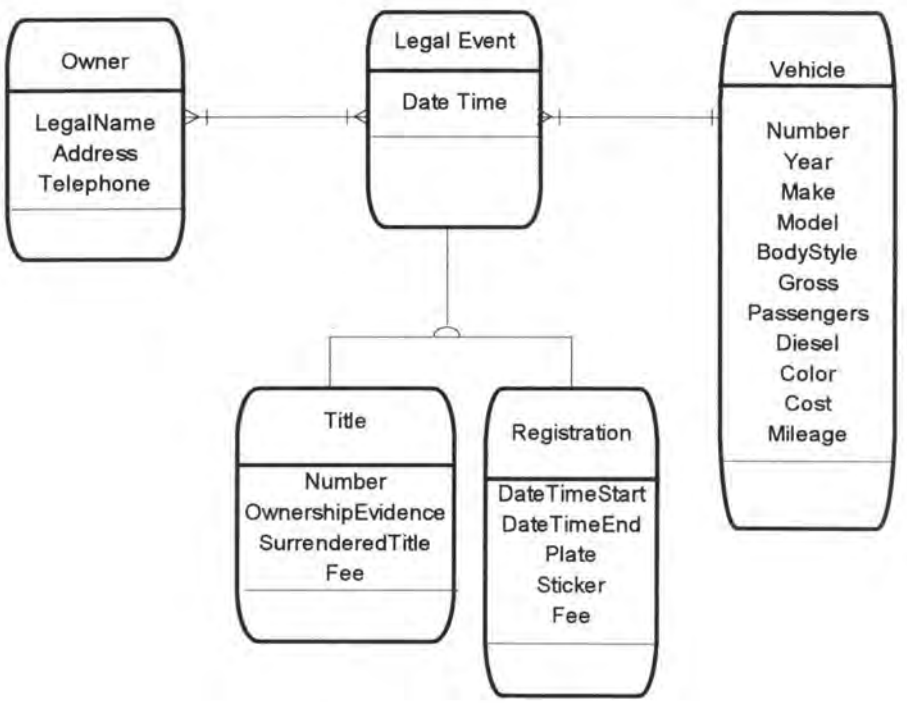


Figure 4-24 : Telos - Contrainte d'intégrité et règle de déduction

4.5 *Coad & Yourdon*

Exemple [COA??].

- Représentation graphique (voir Exemple 4-1).



Exemple 4-1 : Coad & Yourdon - Représentation graphique

- Représentation textuelle (voir Exemple 4-2).

specification legal

definitionAttribute

Legal.

DateTime : the date and time of a legal transaction

Title.

Number : the officially recorded legal transaction number

OwnershipEvidence : the proof of ownership provided

SurrenderedTitle : the origin and number of the surrendered title

Registration.

DateTimeStart : the starting date and time for the registration

DateTimeEnd : the ending date and time for the registration

Plate : the plate issuer, year, type and number

Sticker : the sticker year, type and number

alwaysDerivableAttribute

Legal.

Title.Fee : the fee charged for a title

Registration.Fee : the fee charged for a registration

instanceConnectionConstraint

with Owner 1:M, required

with Vehicle 1:1, required

service Registration.Occur.Add

- will check the arguments against the corresponding Attribute constraints.
- if a violation occurs shall return an error report to the Sender. End.
- will send the message Registration.CalculateFee
- shall output the fees to the Sender

service Registration.AcceptFee

- precondition : immadiately preceded by a no-error Registration.Add
- if the precondition is not met, shall return an error report to the Sender. End.
- will create a new instance of Registration. End.

...

end specification

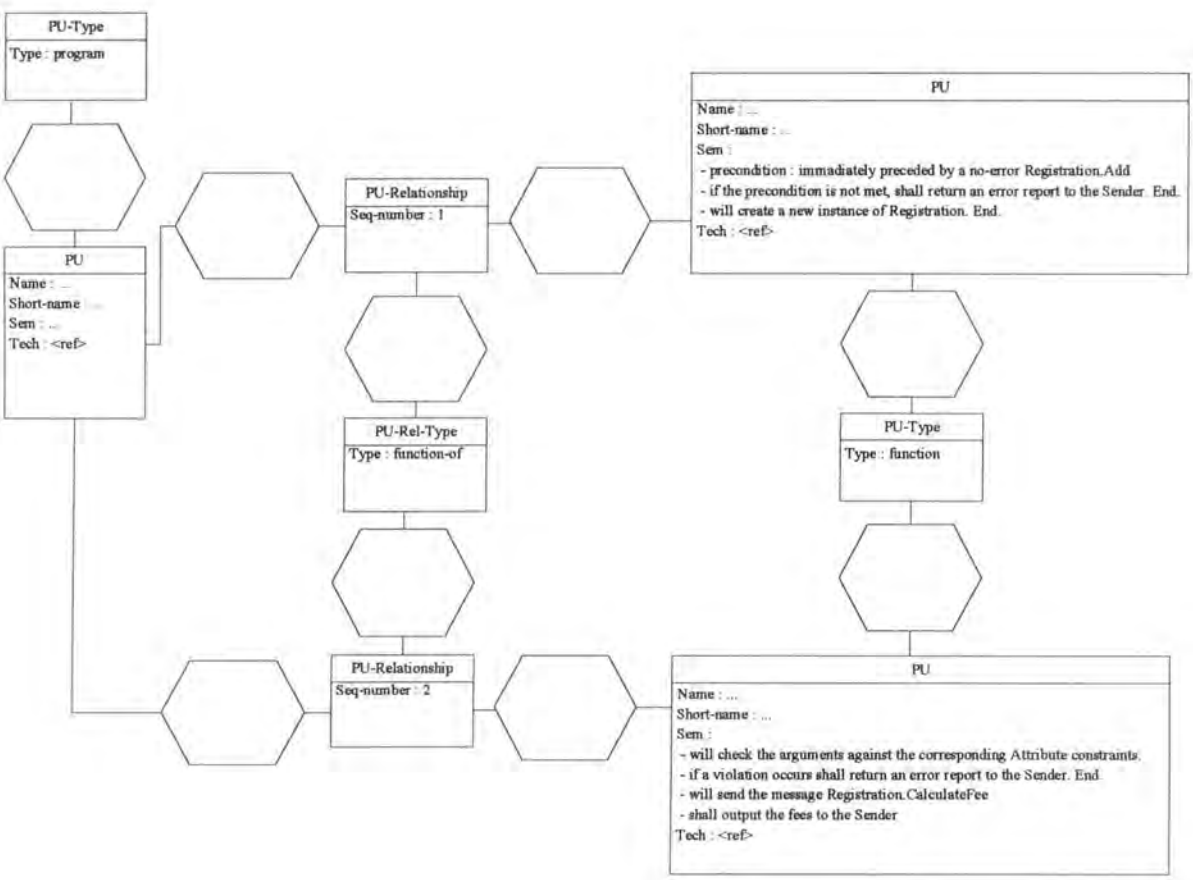
Exemple 4-2 : Coad & Yourdon - Représentation textuelle

Explication.

Cette spécification comprend deux parties : une définition des données et une définition des traitements.

En ce qui concerne la partie données, nous allons créer un *Data-Schema* dans lequel les objets de plus haut niveau seront représentés par des *entity-types*. Ces *entity-types* seront composés d'attributs suivant la décomposition définie dans la spécification. Dans l'attribut *Sem* de l'entité *attribute* seront stockées les descriptions de ces attributs.

Au niveau des traitements, nous créons une *processing unit* par service. Le texte définissant les actions à effectuer sera conservé dans l'attribut *sem*.



4.6 V.D.M.

Dans la Figure 4-25, nous voyons comment la variable club (voir Exemple 2-19 et Exemple 2-20) qui est de type "Systems" (voir Exemple 2-18 et Exemple 2-17) peut être représentée (tous ces exemples sont repris ci-dessous).

Pilots *::*
name : *Names;*
address : *Addresses;*
license : *License;*
account : *Accounts;*
involved : {Yes, No}.

Planes *::*
regnum : *RegNums;*
type : *PlTypes;*
year : *Years;*
triprec : *TriRec;*
hirrate : *HirRates.*

Flights *::*
pilot : *Names;*
plane : *RegNums;*
dt : *Dates;*
tt : *Times;*
dl : *Dates;*
tl : *Times;*
invoice : *Names.*

Systems :: pilots : set of Pilots;
planes : set of Planes;
flights : set of Flights;
where inv-systems = (∀p1, p2 ∈ Pilots . p1 ≠ p2 ⇒ name(p1) ≠ name(p2)) ∧
∀p1, p2 ∈ Planes . p1 ≠ p2 ⇒ regnum(p1) ≠ regnum(p2))

AddPilot (n : Names, a : Addresses, l : Licenses, q : Qualifs)
ext *wr club : Systems*
pre $\forall p \in pilos(club)$. *name(p)* ≠ *n*
post *club'* = μ *(club, pilots* ↦*pilots(club)* ∪ *{mk - Pilots (n, a, l, q, [], yes)})*

ListFlightsPilots (d1, d2 : dates)
lres : c1 :: n : RegNums
f : c2 :: r : RegNums,
dt, dl : Dates,
tt, tl : Times,
in : Names.
ext *rd club : Systems*
pre $d1 \prec_{Dates} d2$
post *lres* = $\{mk - c2(plane(fl), dt(fl), dl(fl), tt(fl), in(fl)) \mid fl \in flights(club) \wedge$
$pilot(fl) - p \wedge ((dt \prec_{Dates} d1 \wedge d2 \prec_{Dates} dl) \vee$
$(d1 \prec_{Dates} dt \wedge dt \prec_{Dates} d2) \vee d1 \prec_{Dates} dl \wedge dl \prec_{Dates} D2))\}\}$

Nous créons une *Entity_type* de nom "club", que nous relions aux trois variables "pilots", "planes" et "flights". Ces variables sont des ensembles. Nous avons choisi de représenter cela en leur attribuant une cardinalité 0-N. Tout attribut de cardinalité 0-N est donc considéré comme un ensemble.

Pour la clarté du schéma, toutes les décompositions n'ont pas été représentées. Seul l'attribut "pilots" a été partiellement décomposé (les attributs "name", "address" et "license" doivent encore être raffinés). Les attributs "planes" et "flights" se décomposent de la même manière.

L'invariant relatif à la variable club sera simplement stocké dans le champ *Technical_desc* (voir Figure 4-25).

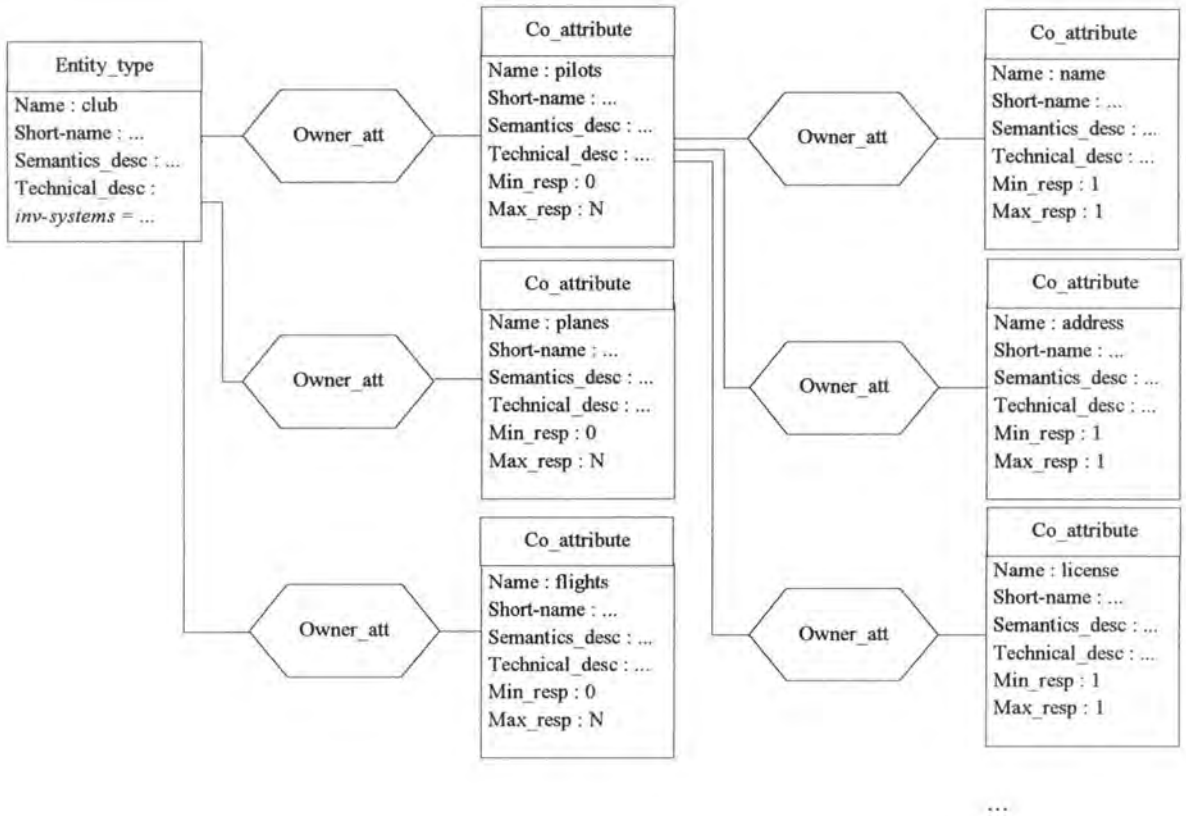


Figure 4-25 : VDM - Représentation d'une variable composée

Chaque opération (voir Exemple 2-19) est ensuite représentée par une *Processing Unit*. Ces *Processing Units* sont alors reliées à leurs différents paramètres (en entrée et en sortie). Elles sont également décomposées en deux parties représentant les pré et post-conditions. La ligne indiquant s'il s'agit d'un d'accès en lecture ou en écriture est stockée dans le champs *Tech*.
Nous pouvons visualiser cela sur le schéma suivant (voir Figure 4-26).

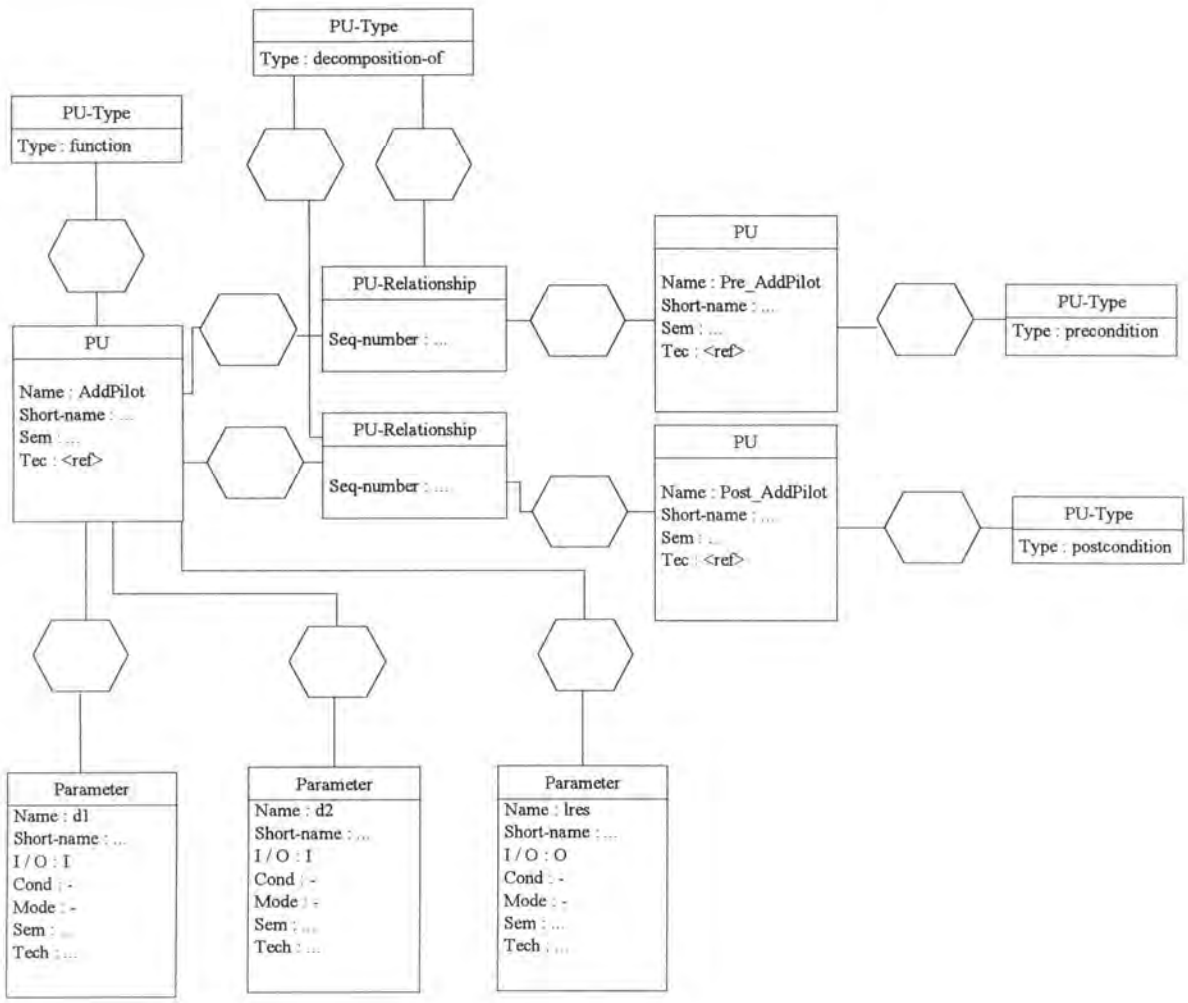
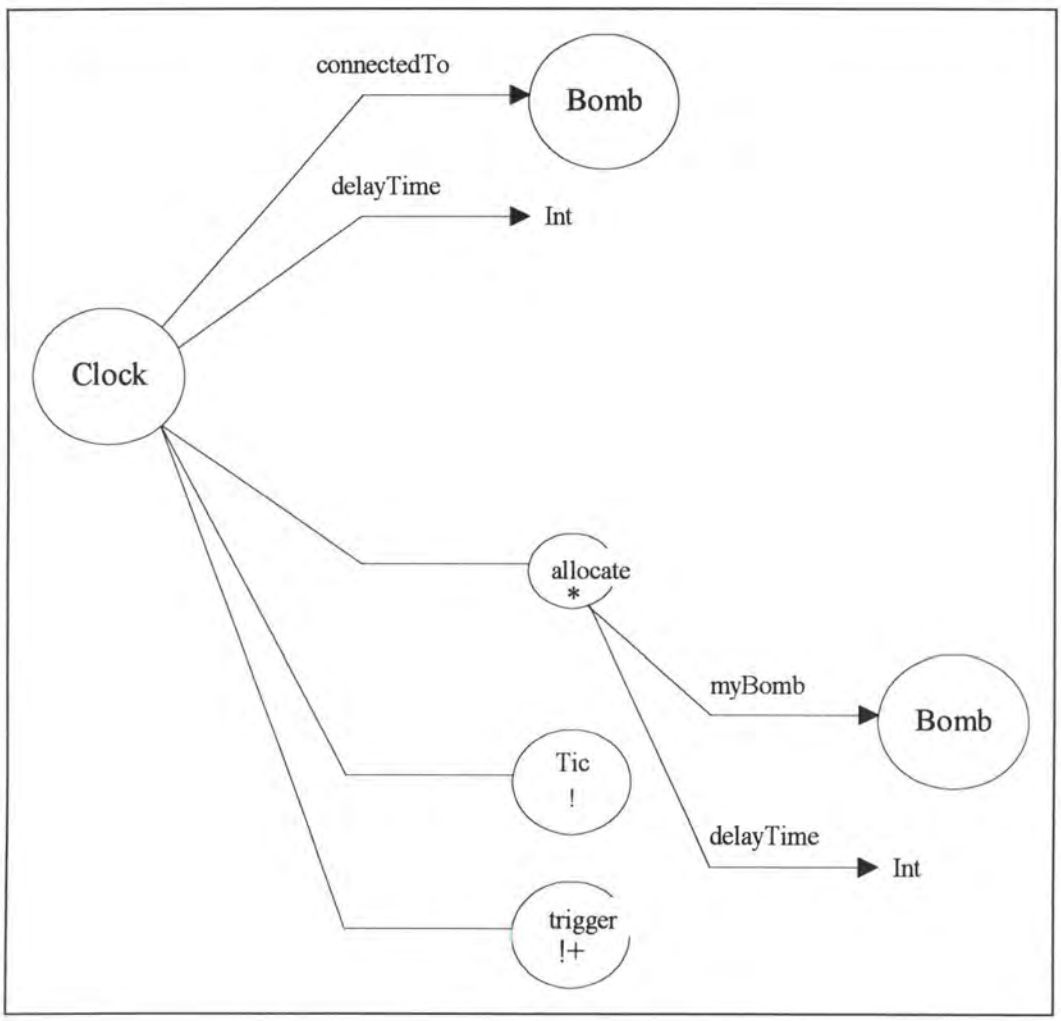


Figure 4-26 : VDM - Représentation d'une opération avec paramètres

Il est bien sûr évident que toutes ces entités *Parameter* doivent être reliées à leur *Entity_type* correspondant. Cela n'a pas été représenté pour des raisons de clarté.

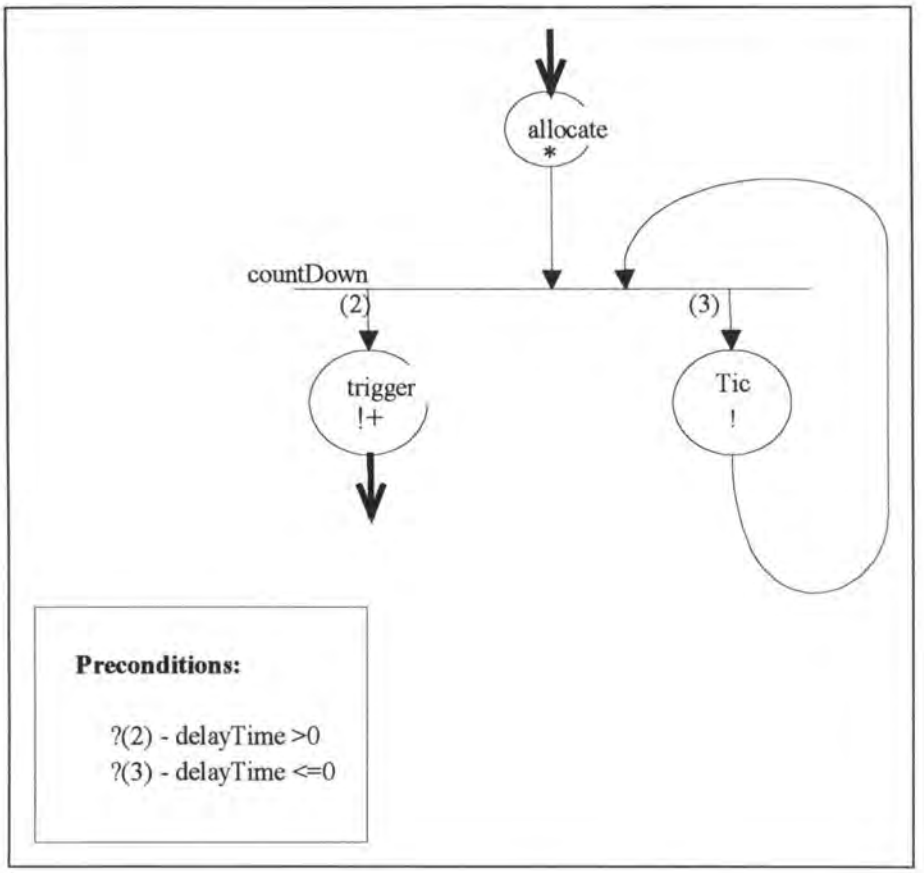
4.7 Oblog

Exemple[8]



Exemple 4-3 : OBLOG - Schéma de définition

[8] Cet exemple est tiré de [ZEI95].



Exemple 4-4 : OBLOG - Schéma de comportement

Dans cet exemple (voir Exemple 4-3 et Exemple 4-4), l'objet dans son ensemble sera représenté par une *entity_type*. Ses attributs seront modélisés par des *attributes* dans DB-Main (voir Figure 4-27).

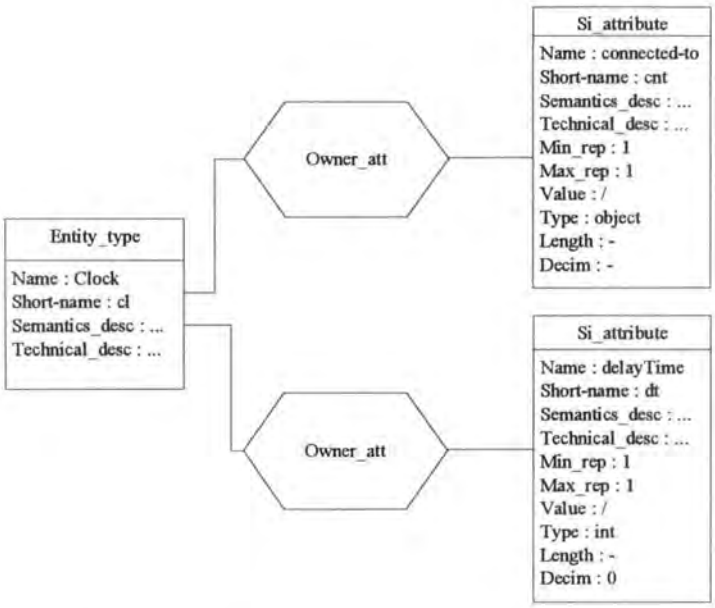


Figure 4-27 : OBLOG - Modélisation de la partie données de l'objet

Les traitements relatifs à l'objet seront, quant à eux, représentés chacun par une *processing unit*. La *processing unit* est reliée à une *entity_type* via l'entité *parameter* (voir Figure 4-28).

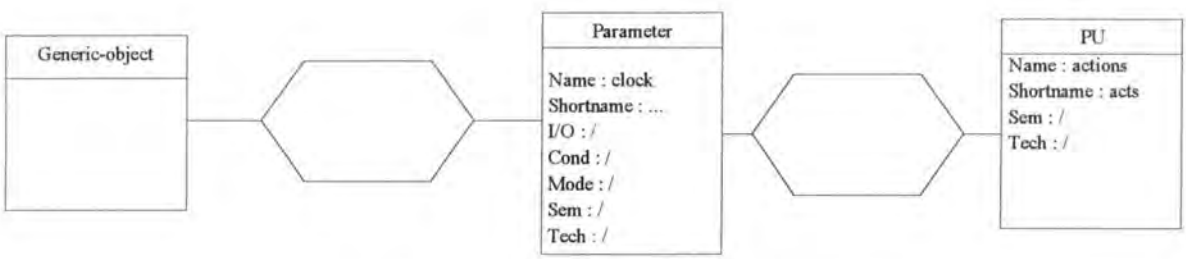


Figure 4-28 : OBLOG - Modélisation de la partie traitements de l'objet

Le schéma de comportement de l'objet nous fait créer des relations entre les différentes *processing units*. Dans l'attribut *sem* de la *processing unit* sera stockée la condition d'activation de la fonction (voir Figure 4-29).

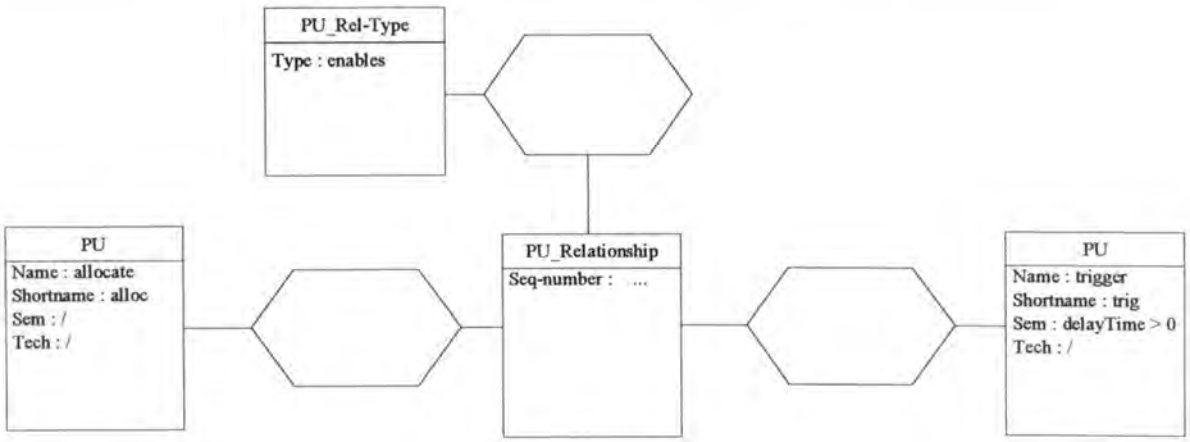


Figure 4-29 : OBLOG - Modélisation d'une séquence d'actions

Les actions d'un objet modifient les attributs. Ce sont en quelque sorte des instructions qui pourront donc être représentées par des *processing units*. Ces *processing units* seront reliées à l'action par une *PU-Relationship* (voir Figure 4-30).

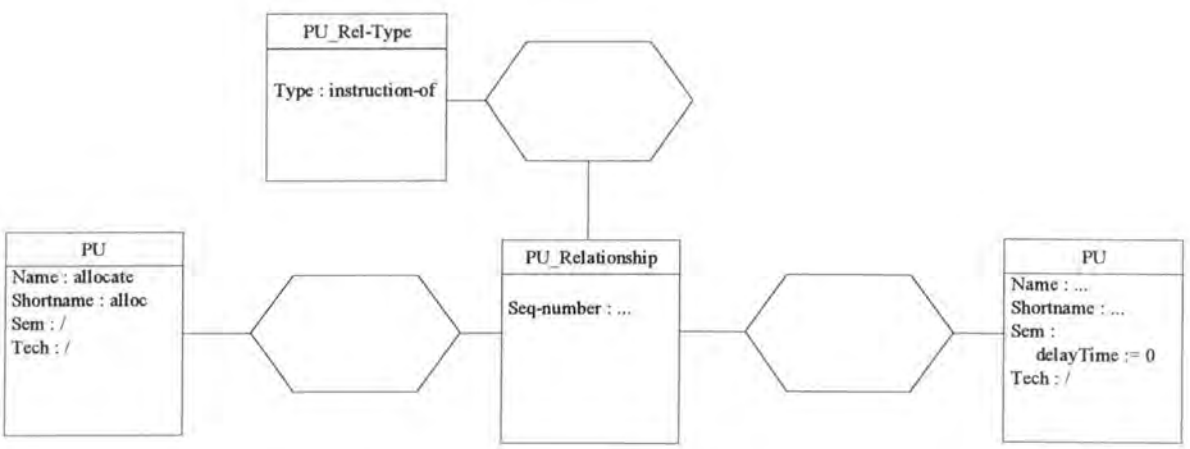


Figure 4-30 : OBLOG - Décomposition d'une action en instructions

4.8 Le langage C

Nous allons montrer ici comment un programme C peut s'insérer dans le modèle que nous avons proposé.

a) Un programme

```
#include <stdio.h>

main ()
{
        ...
}
```

Exemple 4-5 : C - Un programme

Le fichier contenant le programme (voir Exemple 4-5) ainsi que le fichier *stdio.h* constitueront un *PU_Schema*. Les noms des fichiers composant ce *PU_Schema* seront gardés dans l'attribut *tech*. A partir du *PU_Schema*, on vérifiera s'il existe un ou plusieurs programmes en examinant les clauses *#include* et une *processing unit* sera créée par programme. Comme il n'existe ici qu'un seul programme, une seule *processing unit* sera créée. Ce programme définit plusieurs variables et les utilise toutes. La *processing unit* correspondante sera donc reliée à deux entités *parameter* représentant le schéma des données persistantes et le schéma des données non persistantes. Ces entités *parameter* seront, à leur tour, reliées aux deux schémas représentés ici par leur *Generic-Object* (voir Figure 4-31).

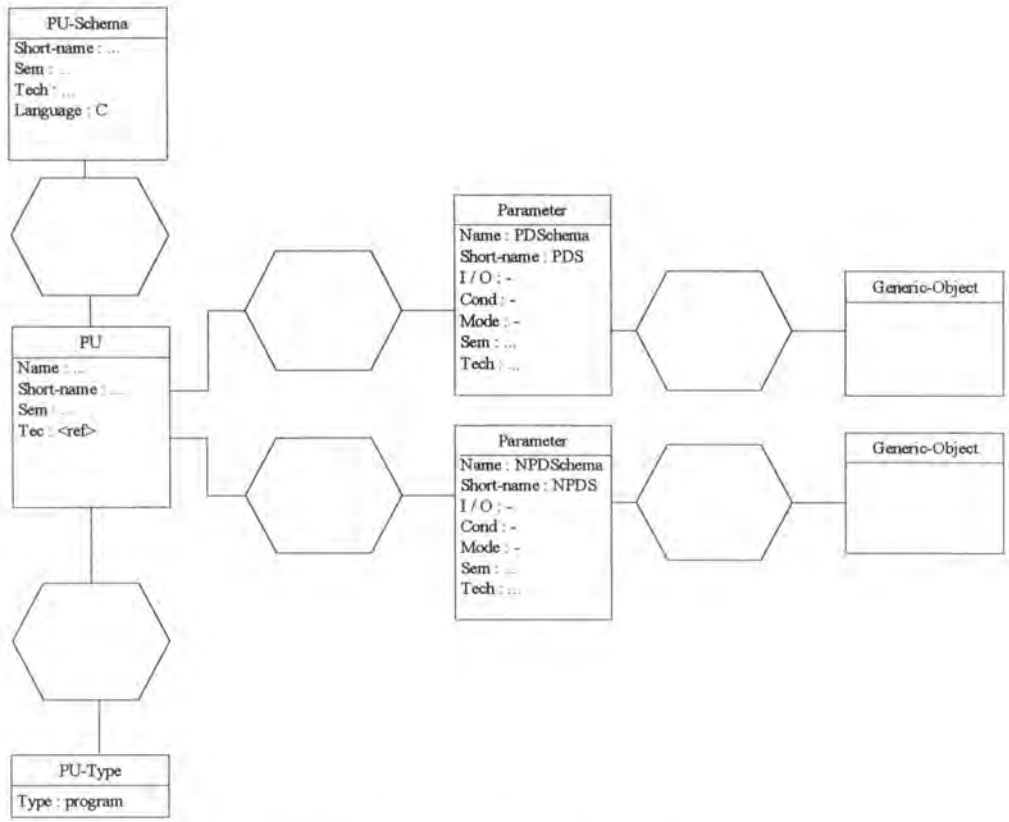


Figure 4-31 : C - Modélisation d'un programme

b) Un fichier

Un programme peut être composé de plusieurs fichiers source. Chaque fichier sera représenté par une *processing unit*. Une entité *PU_relationship* sera créée pour symboliser le lien existant entre le programme et chacun de ses modules. Cette entité *PU_Relationship* indiquera le numéro de séquence correspondant à la décomposition de la *processing unit*. L'attribut *type* de l'entité *PU-Rel-Type* représente le type de relation entre les deux *processing units* (voir Figure 4-32).

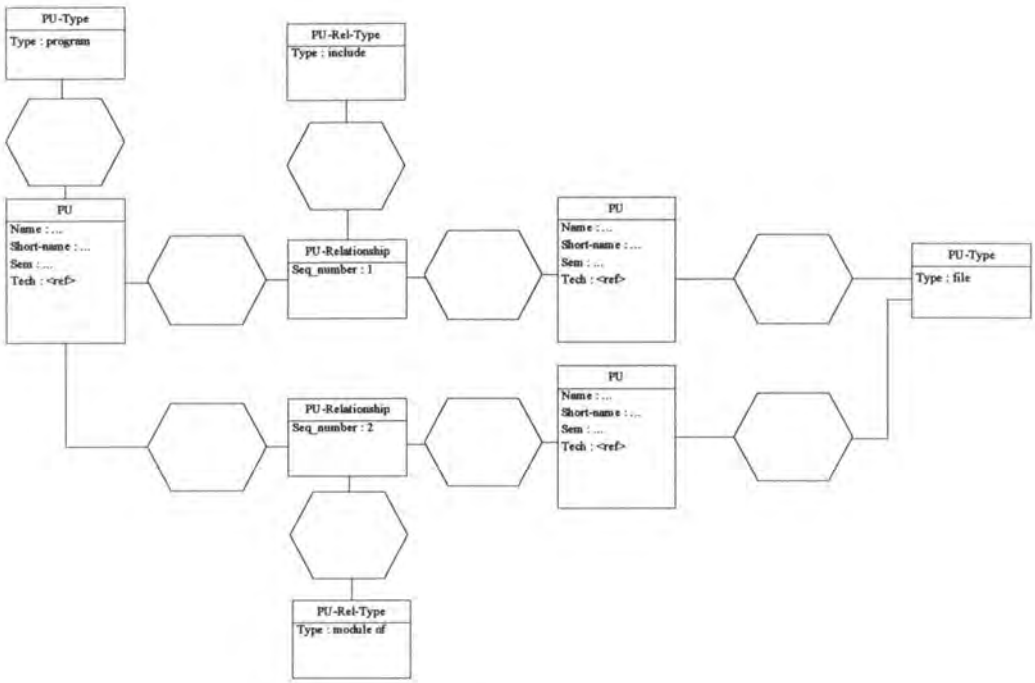


Figure 4-32 : C - Décomposition d'un programme en fichiers

c) Une fonction

```
int f (int x)
{
        ...
}
```

Exemple 4-6 : C - Une fonction

Un module est décomposé en fonctions (voir Exemple 4-6). Chaque fonction constituera une *processing unit*. Cette *processing unit* sera reliée au module via *PU_Relationship* toujours en indiquant le type de relation entre les deux *processing units*. Pour chaque paramètre de la fonction, une entité *parameter* sera créée. Elle précisera le nom des paramètres, leur type et leur mode (passage par adresse ou par valeur). On indiquera également si ces paramètres sont des inputs ou des outputs. Ces entités *parameter* seront alors reliées aux *generic-objects* correspondants (voir Figure 4-33).

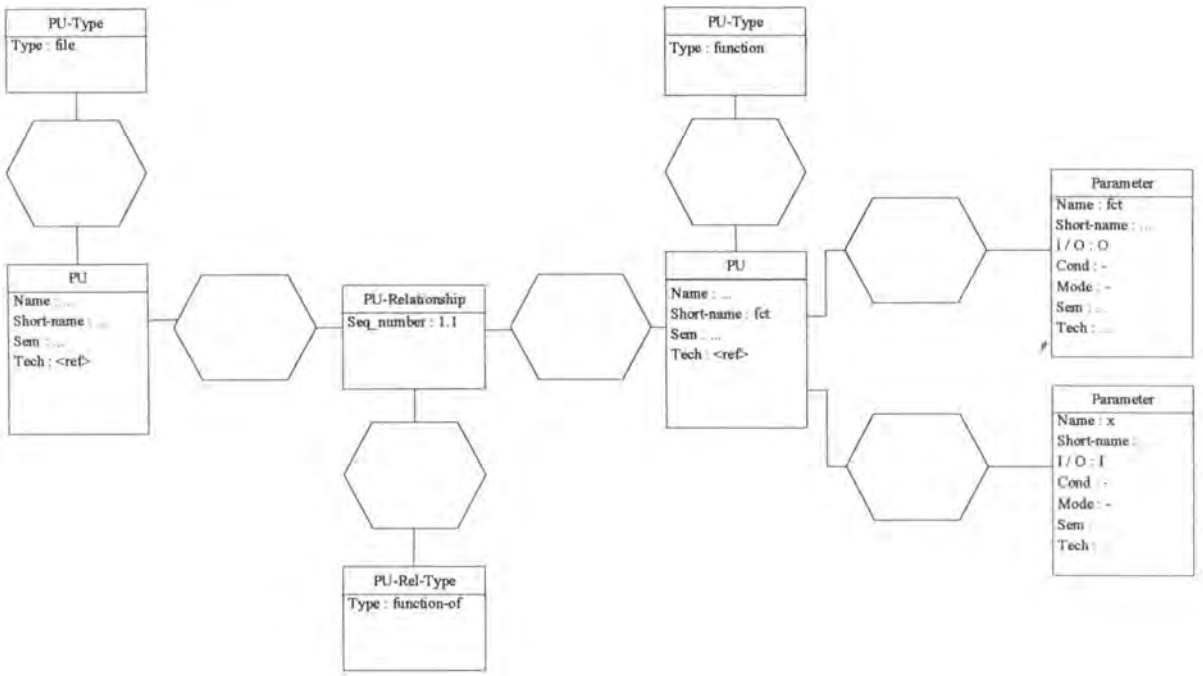


Figure 4-33 : C - Décomposition d'un fichier en fonctions

d) Une instruction

Une instruction sera, elle aussi, représentée par une *processing unit*. Selon le type de l'instruction, la *processing unit* sera ou non décomposée. En outre, lorsqu'une instruction fait référence à une variable, un lien est créé entre la *processing unit* et cette variable représentée par un *parameter*. Ce *parameter* sera ensuite relié à l'objet adéquat.

L'instruction d'affectation

```
a = b + 100
```

Exemple 4-7 : C - Une instruction d'affectation

Nous voyons deux possibilités de représentation d'une instruction d'affectation (voir Exemple 4-7). Chacune d'entre elles possède ses avantages et ses inconvénients. Nous allons les présenter toutes les deux.

La première représentation que nous proposons nous fait créer une *processing unit* afin de modéliser l'instruction. La *processing unit* sera ensuite reliée à ses *parameters* et ce, de la façon suivante : quatre entités *parameter* sont créées. Elles représentent les inputs (affectants) et les outputs (affectés) et indiquent de plus leur nom ainsi que leur type (voir Figure 4-34).

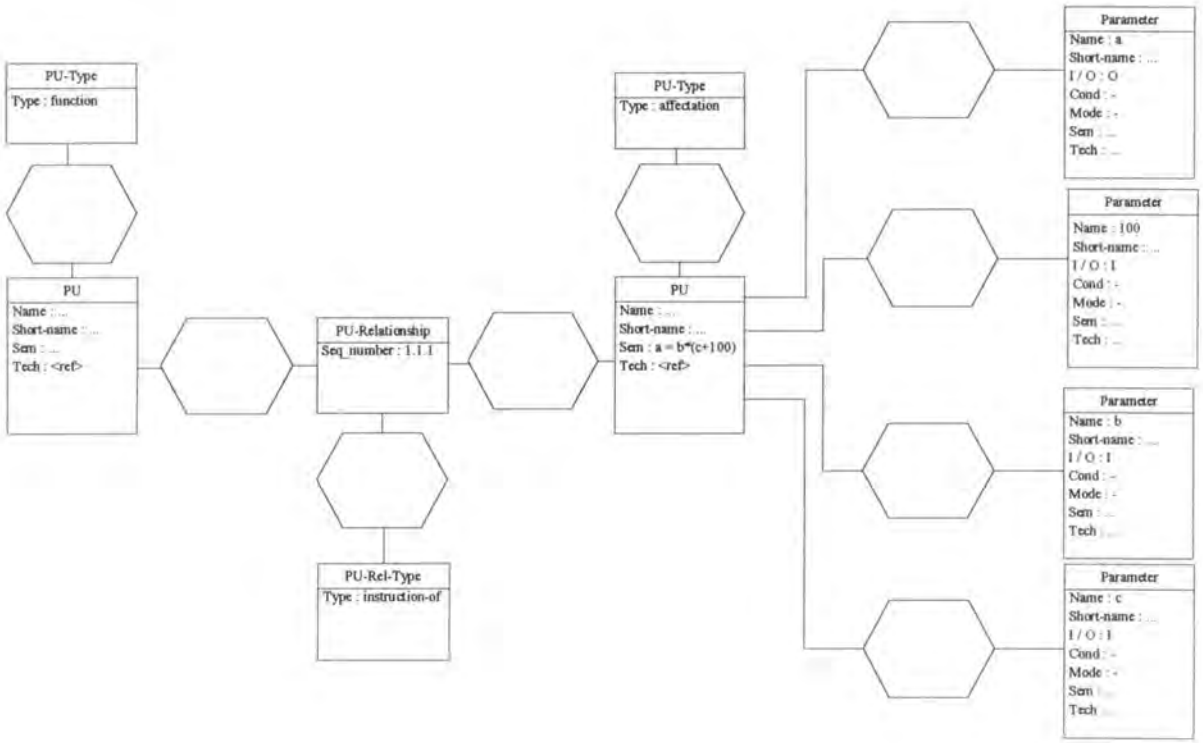


Figure 4-34 : C - Modélisation d'une instruction d'affectation

Cette représentation a comme avantage d'être très compacte mais présente un gros désavantage, celui d'être dépendante du langage utilisé. En effet, si l'on veut pouvoir, à partir de notre modèle, faire une traduction du programme d'un langage à un autre, il faudra tenir compte du langage initial.

La deuxième représentation possible, par contre, est indépendante du langage mais, en contrepartie, consomme énormément d'espace de stockage. Elle consiste à décomposer l'instruction en créant d'autres *processing units*. Nous créons en fait l'arbre syntaxique de l'instruction.

Le schéma suivant (voir Figure 4-35) montre la décomposition de l'instruction en deux parties : affectant et affecté.

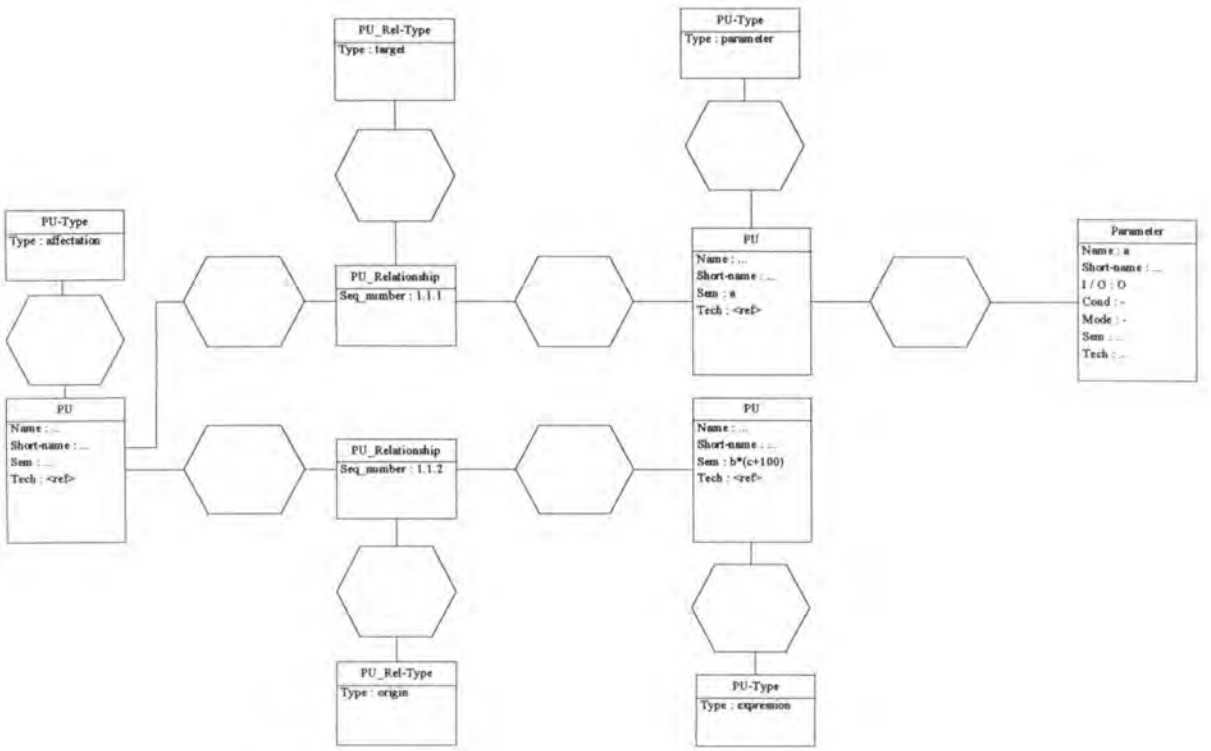


Figure 4-35 : C - Modélisation d'une instruction d'affectation

Le schéma suivant (voir Figure 4-36) reprend la *processing unit* représentant l'affectant et la décompose encore un peu plus.

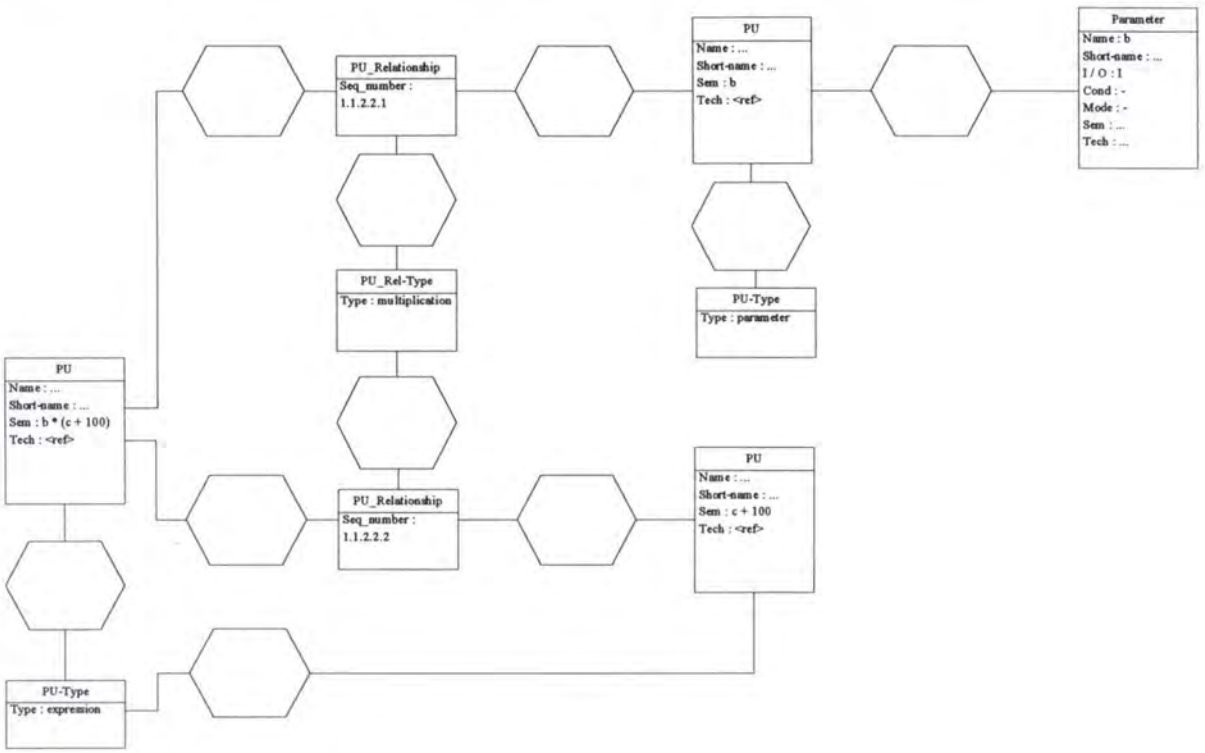


Figure 4-36 : C - Modélisation d'une instruction d'affectation

C'est ensuite au tour de la *processing unit* représentant c + 100 d'être décomposée afin de modéliser un seul paramètre (voir Figure 4-37).

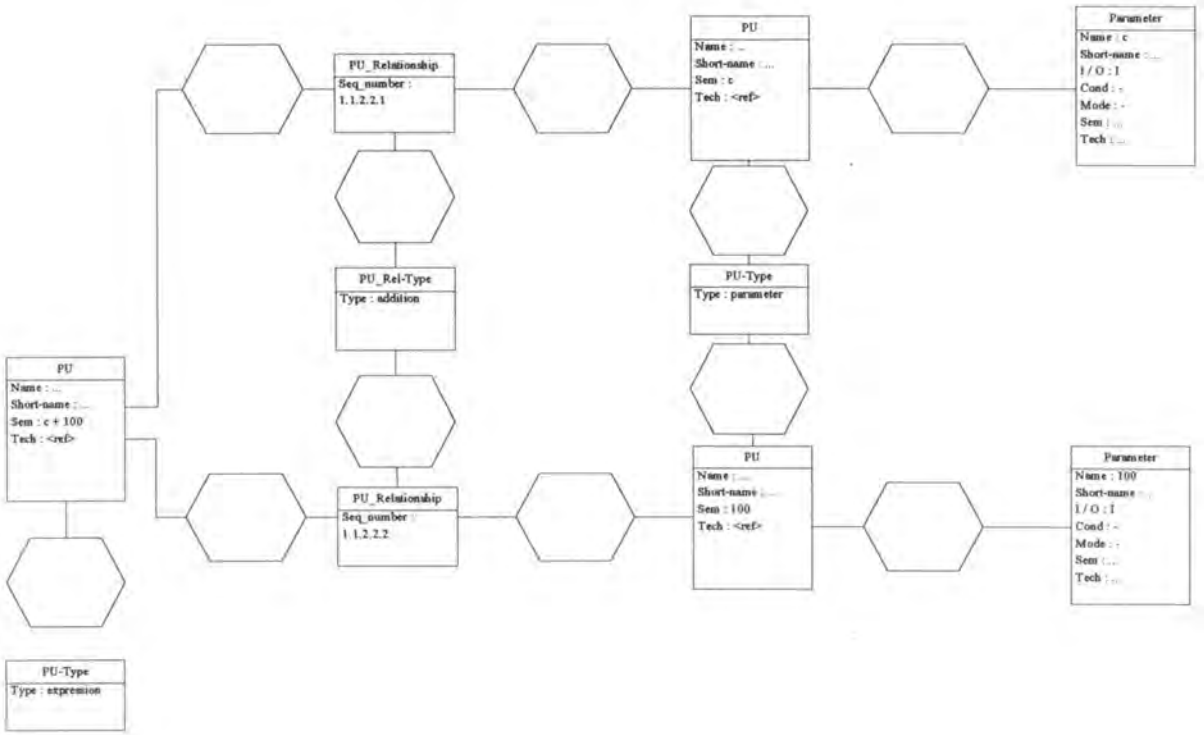


Figure 4-37 : C - Modélisation d'une instruction d'affectation

Tout comme auparavant, les entités *parameter* seront reliées aux objets qu'elles représentent.

L'instruction if ... else

```
if (i > 100)
{
        ...
}
else
{
        ...
}
```

Exemple 4-8 : C - Une instruction if ... else

L'instruction if ... else (voir Exemple 4-8), contrairement à l'instruction d'affectation, n'est pas une instruction simple. Elle sera donc décomposée en deux ou trois *processing units*

(représentant la partie condition, la partie then et, si cette dernière est présente, la partie else de l'instruction). Ces deux ou trois *processing units* seront reliées à la première via *PU_Relationship* en précisant le type de relation (condition, then ou else).

Les variables intervenant dans la condition seront des paramètres de la *processing unit* de type condition. Les parties then et else de l'instruction seront ensuite, si nécessaires décomposées en instructions plus simples et reliées à leurs paramètres comme précédemment (voir Figure 4-38).

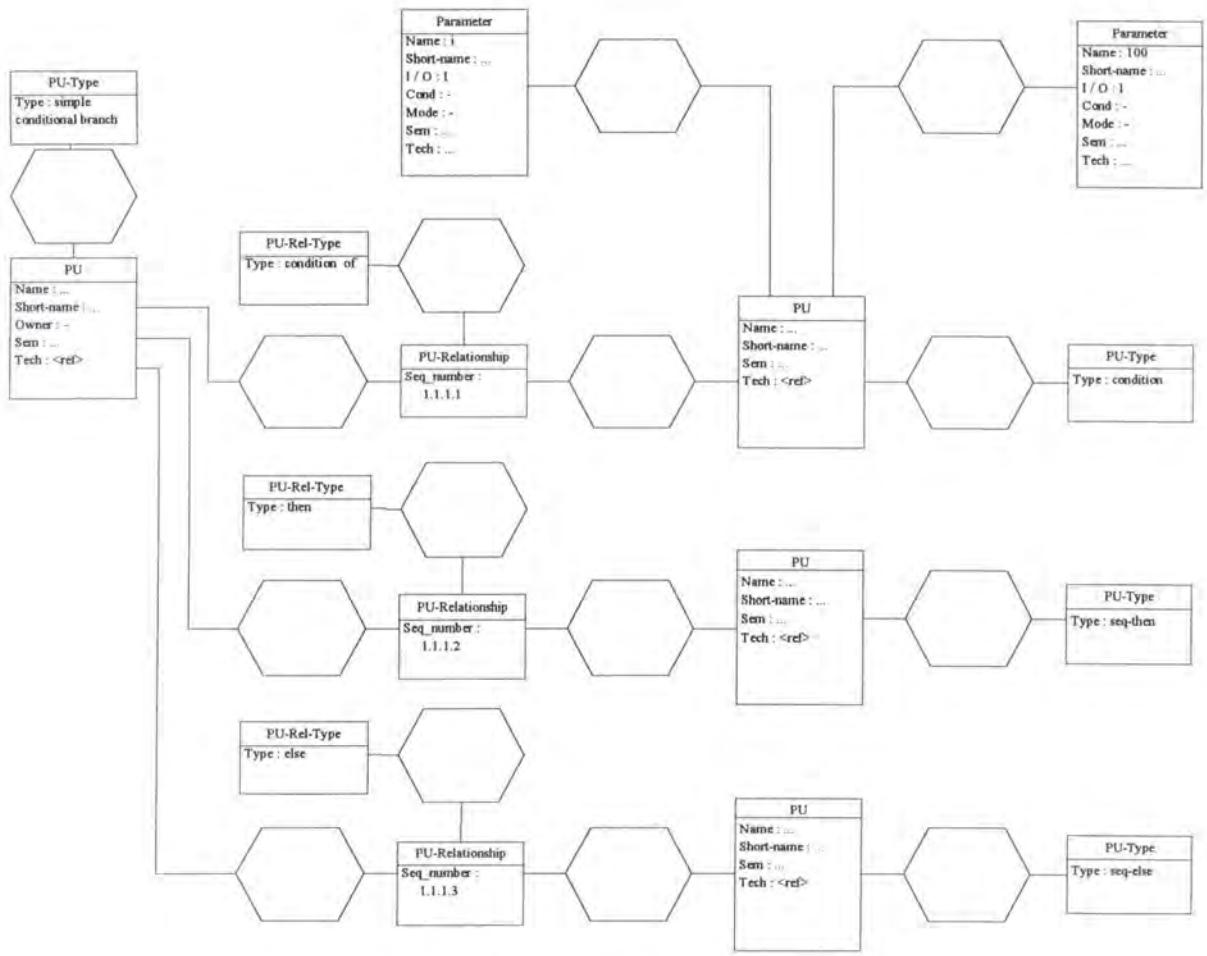


Figure 4-38 : C - Modélisation d'une instruction if ... else

L'instruction switch

```
switch (i)
{
        case 1 :
                ... ;
                break ;
        case 2 :
                ... ;
                break ;
        default :
                ... ;
}
```

Exemple 4-9 : C - Une instruction switch

Tout comme pour l'instruction if ... else, l'instruction switch (voir Exemple 4-9) fera l'objet d'une décomposition supplémentaire. Deux *processing units* seront créées. La première représentant la condition du switch et la deuxième symbolisant le bloc d'instructions à exécuter si cette première condition est vérifiée. Une troisième *processing unit* sera ensuite créée. Elle représentera le reste de l'instruction switch. Cette *processing unit* sera, par la suite, traitée comme un switch normal.

La liaison entre l'instruction switch et les paramètres se fera de la même façon que pour l'instruction if ... else (voir Figure 4-39).

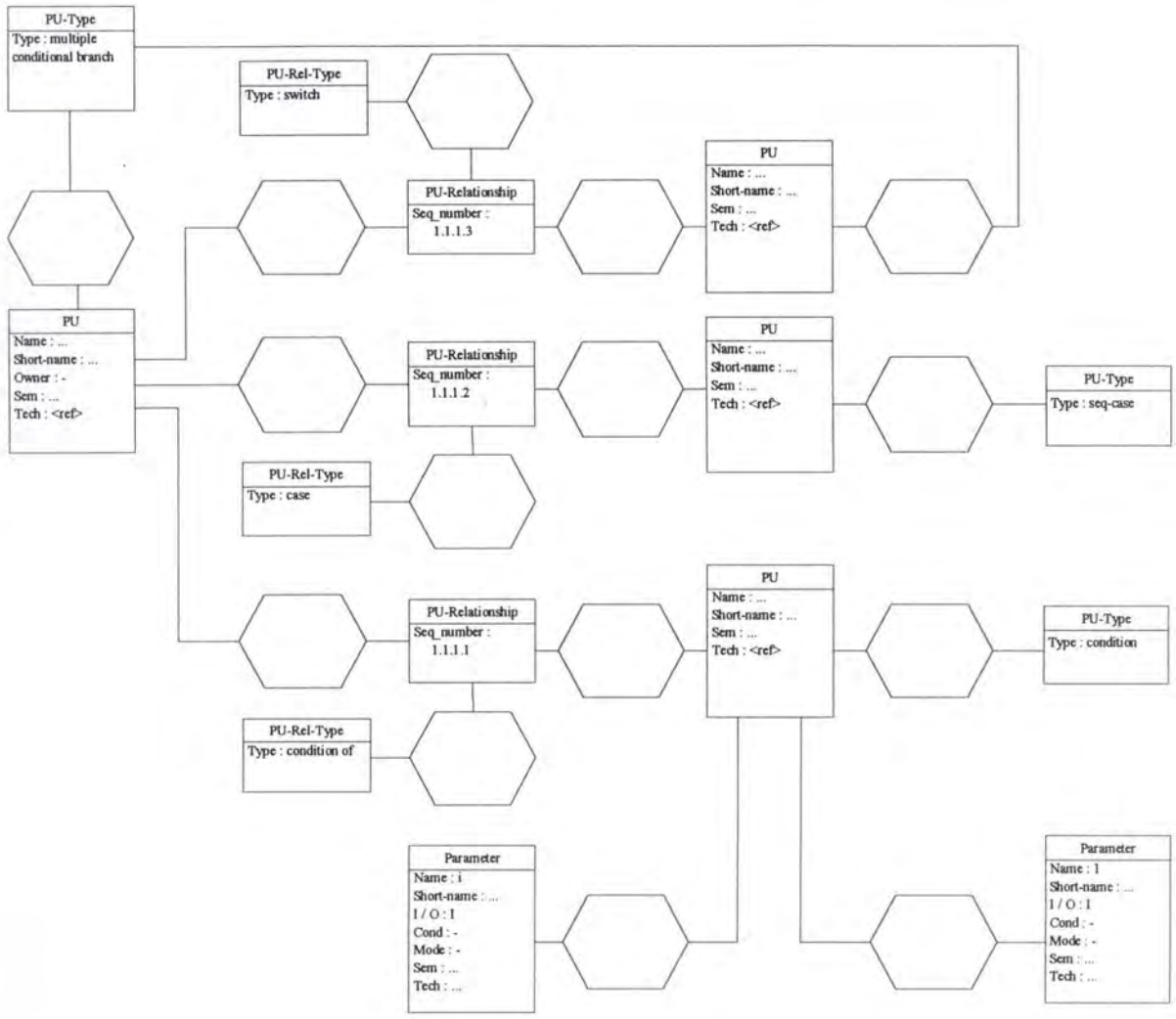


Figure 4-39 : C - Modélisation d'une instruction switch

L'instruction while

```
while (i < 100)
{
        ...
}
```

Exemple 4-10 : C - Une instruction while

La boucle en elle-même (voir Exemple 4-10) constitue une *processing unit*. Il en sera de même pour le contenu de la boucle ainsi que pour la condition de la boucle. Ces deux *processing units* seront reliés par une *PU_Relationship*. L'entité *PU_Type* indiquera le type de la boucle.

Le lien entre la *processing unit* while et les objets utilisés se fait de la même manière que pour l'instruction if ... else (voir Figure 4-40).

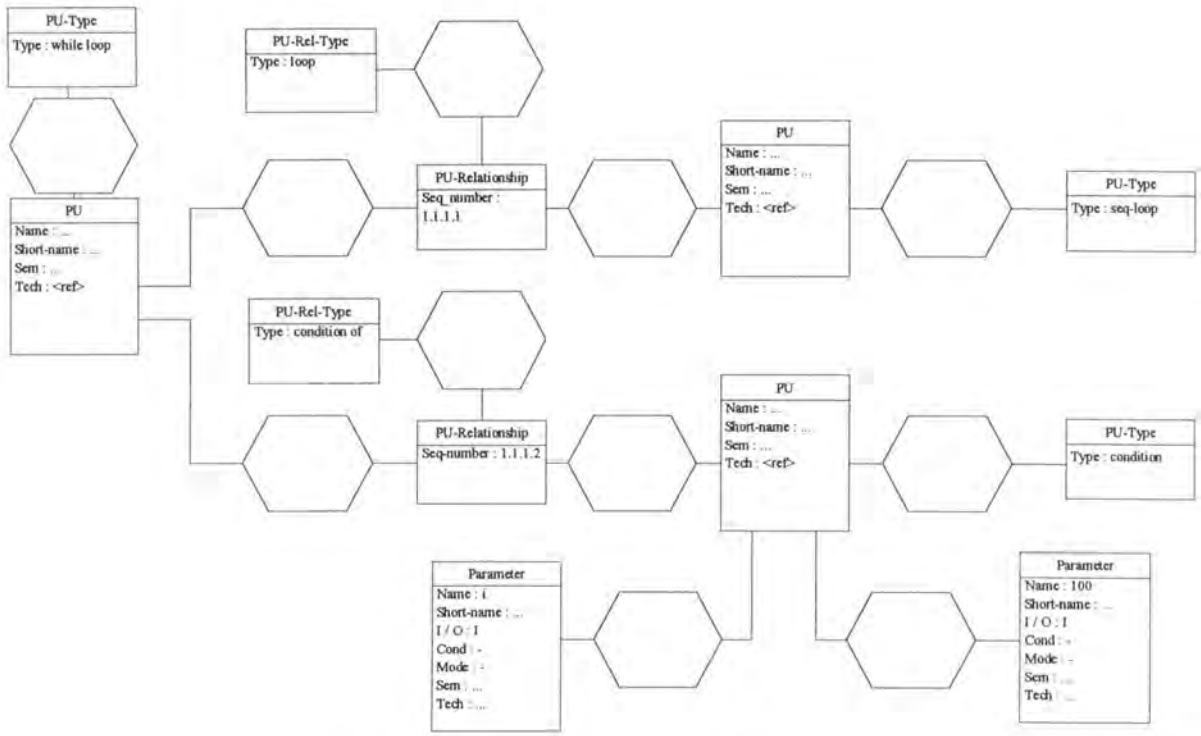


Figure 4-40 : C - Modélisation d'une instruction while

L'instruction do ... while

```
do
{
        ...
}
while (i < 100)
```

Exemple 4-11 : C - Une instruction do ... while

La seule différence entre cette instruction (voir Exemple 4-11) et la précédente est la valeur de l'attribut *type* de l'entité *PU_Type* (voir Figure 4-41).

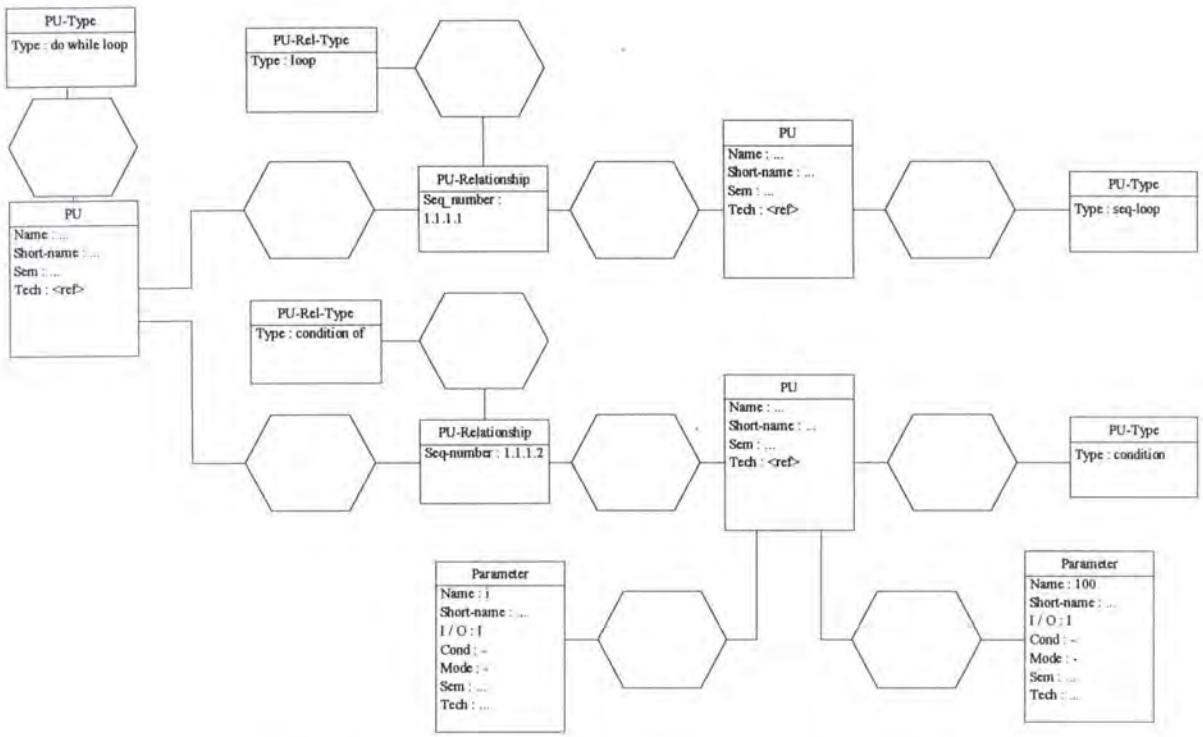


Figure 4-41 : C - Modèlisation d'une instruction do ... while

L'instruction for

```
for (i = 1, i < 100, ++i)
{
        ...
}
```

Exemple 4-12 : C -Une instruction for

Cette forme de boucle (voir Exemple 4-12) est un cas assez particulier. En effet, dans la définition même de la boucle se trouvent l'initialisation du compteur, la condition et l'incrémentation du compteur. Voici la solution que nous proposons. Comme d'habitude la boucle en elle-même ainsi que la condition constitueront chacune une *processing unit*. Il en sera de même pour l'initialisation du compteur et pour l'incrémentation de ce même compteur.

La liaison avec les *generic_objects* se fera de la même façon que pour l'instruction if ... else (voir Figure 4-42).

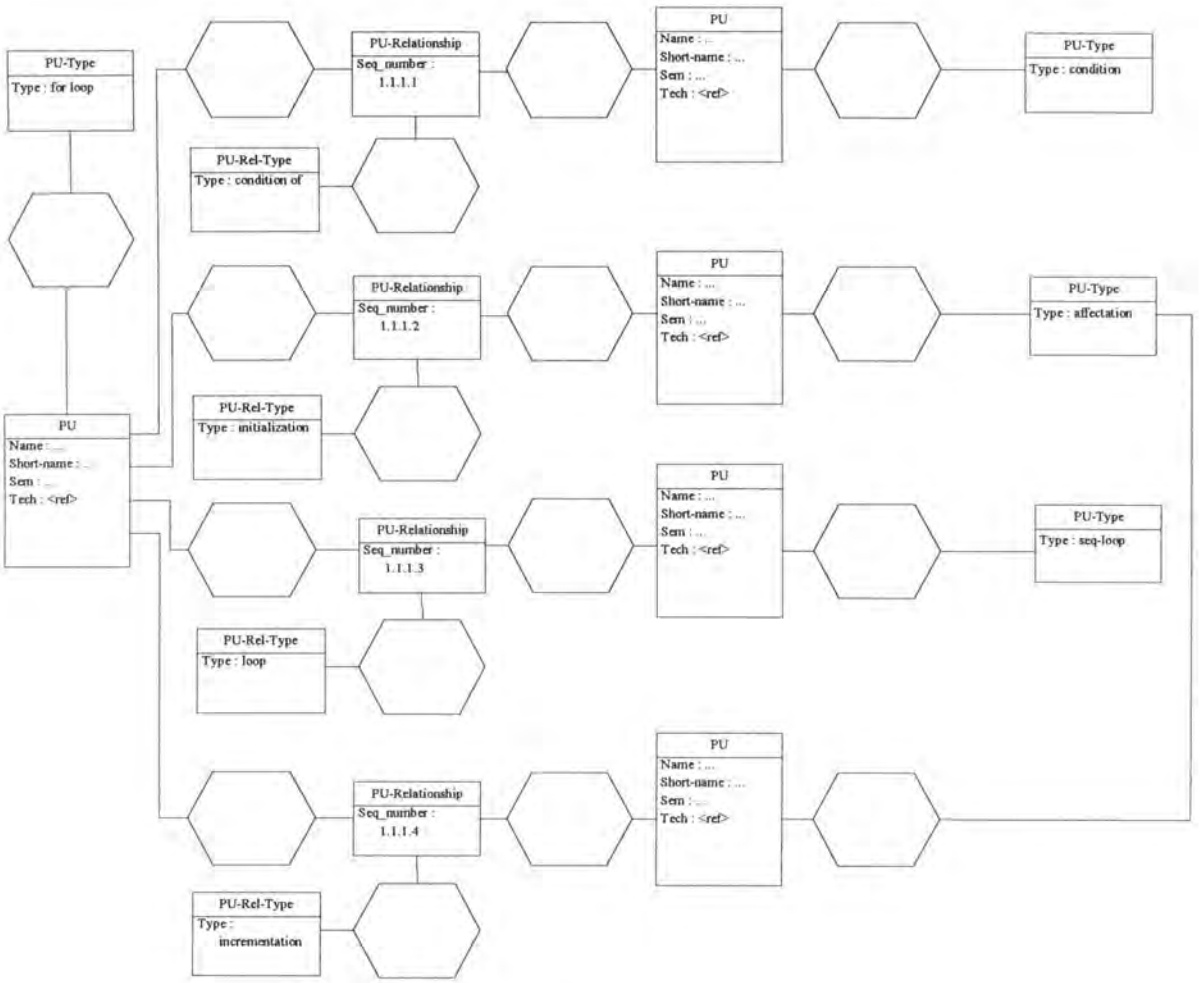


Figure 4-42 : C - Modélisation d'une instruction for

L'appel de fonction

Un appel de fonction sera représenté par une *processing unit*. Si un des paramètres de la fonction est un appel de fonction, une nouvelle *processing unit* sera créée. Les paramètres simples seront, eux, représentés par des entités *parameter*.

Ensuite, la *processing unit* de type « function call » sera reliée à la *processing unit* représentant la définition de la fonction (voir Figure 4-43).

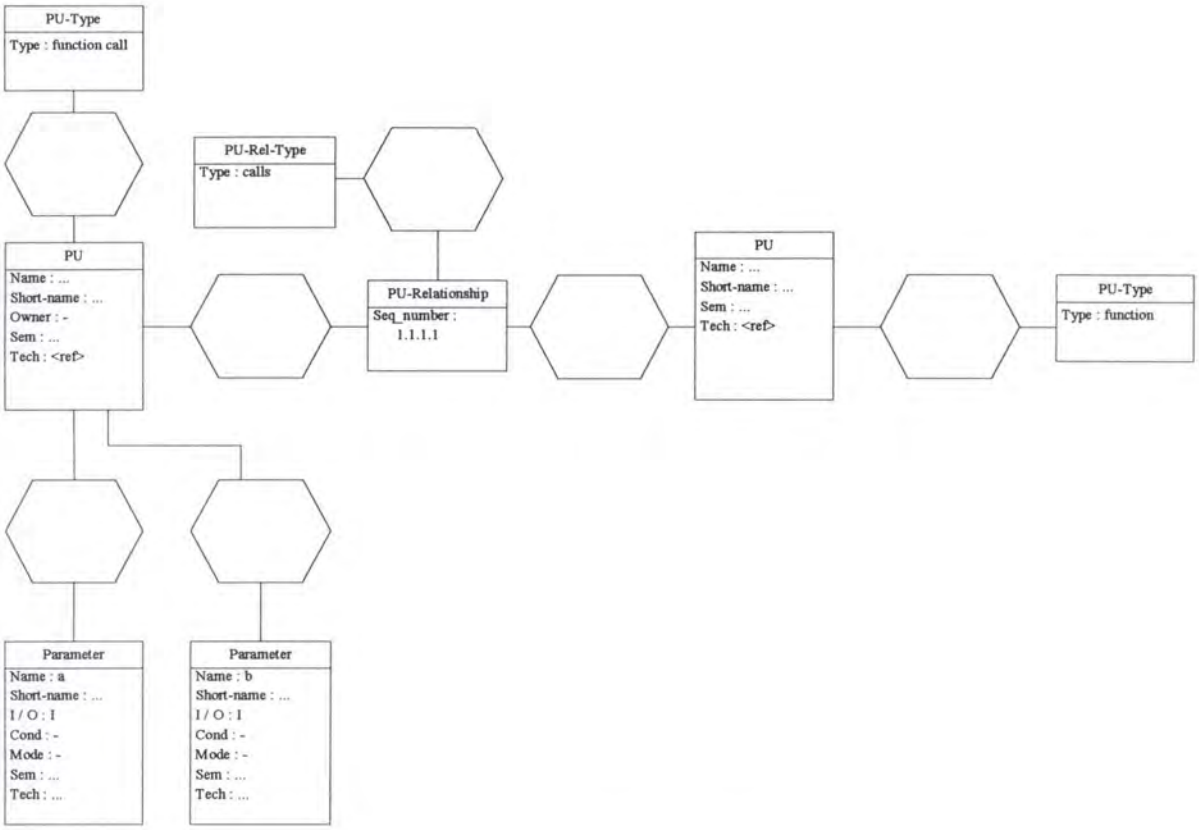


Figure 4-43 : C - Modélisation d'un appel de fonction

4.9 Le langage C++

Pour montrer comment un programme orienté-objet peut s'insérer dans le repository de DB-Main, nous avons choisi d'analyser quelques structures typiques du C++. Nous allons donc analyser la notion de classe et d'héritage sur les classes, la notion de contrôle d'accès aux éléments d'une classe, ainsi que la notion de surcharge de fonctions et d'opérateurs.

Notion d'héritage

Exemple [ACH93]

```
...
class moyen-transport
{
...
};
class vehicule-terrestre : moyen-transport
{
...
};
class bateau : moyen-transport
{
...
};
class avion : moyen-transport
{
...
};
class voiture : véhicule terrestre
{
```

```
...
};
class camion : véhicule terrestre
{
...
};
```

Exemple 4-13 : C++ - Définition de classes

Explication

La classe (voir Exemple 4-13) constitue une *entity-type*. Elle contiendra comme attributs les attributs définis dans la classe. La partie traitement de la définition sera représentée par une *processing unit*. Les méthodes seront alors traitées de la même façon que des procédures conventionnelles (voir Figure 4-44 et Figure 4-45).

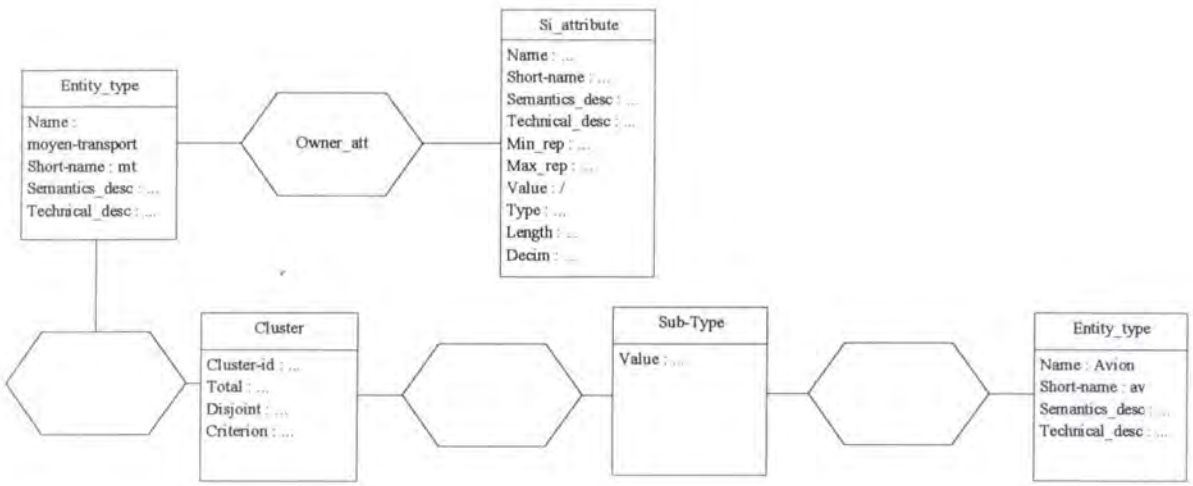


Figure 4-44 : C++ - Modélisation de l'héritage entre classes

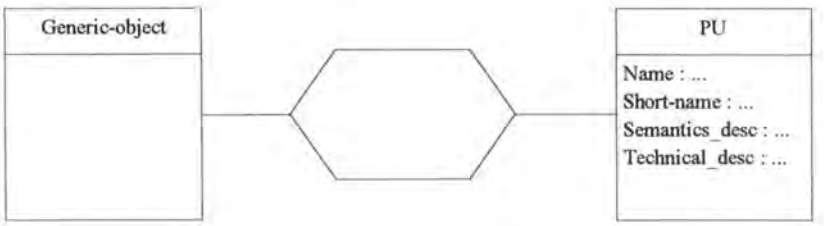


Figure 4-45 : C++ - Lien entre la partie traitements et la partie données

Contrôle d'accès à une classe

Exemple

```
Class AccessControlExample
{
        int value_1;
        void f_1(long);

private :
        int value_2;
        int f_2(char *);

public :
        char* value_3;
        long f_3();

protected :
        int value_4;
        void f_4(long);
};
```

Exemple 4-14 : C++ - Définition d'une classe

Explication

Dans l'attribut *tech* de l'attribut ou de la *processing unit* (voir Exemple 4-14), nous indiquerons une clause du genre *access = private* indiquant le mode d'accès à l'objet considéré.

Surcharge

1) Surcharge des fonctions

Exemple

```
int f1(int i)
{
...
};
int f1(char c)
{
...
};
main()
{
        f1(0);          // Appel avec argument int
        f1('a');        // Appel avec argument char
}
```

Exemple 4-15 : C++ - Définition d'une fonction surchargée

Explication

Dans ce cas (voir Exemple 4-15), nous allons créer deux *processing units* de type function. Ces deux fonctions seront reliées à leurs paramètres respectifs en entrée et en sortie. Les entités *parameter* seront ensuite reliées à leur *generic-object* (voir Figure 4-46).

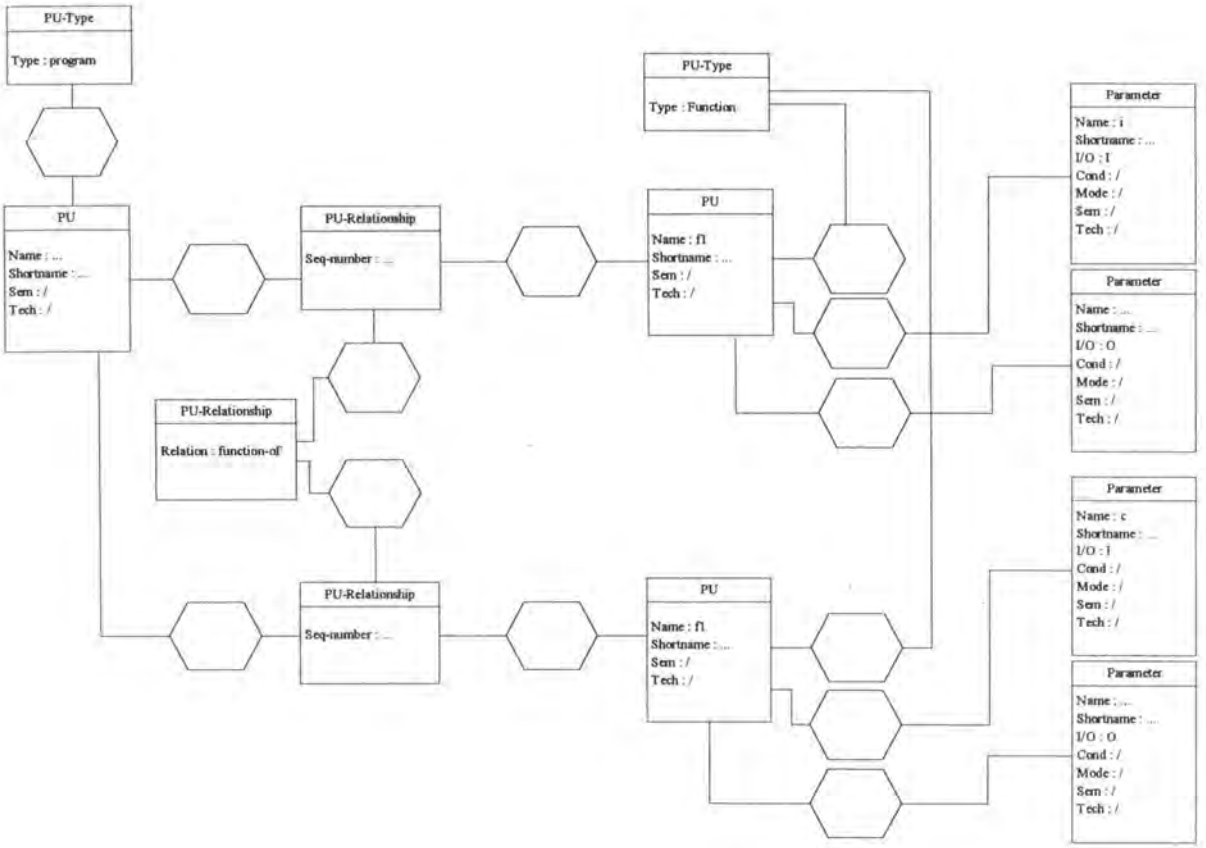


Figure 4-46 : C++ - Modélisation d'une fonction surchargée

Lors de l'appel d'une de ces fonctions, nous créons une *processing unit* contenant cette instruction. Cette *processing unit* sera de type *Function-call*. Nous relions cette *processing unit* à la *processing unit* fonction correspondante, ainsi qu'à ses différents paramètres en entrée et en sortie (voir Figure 4-47).

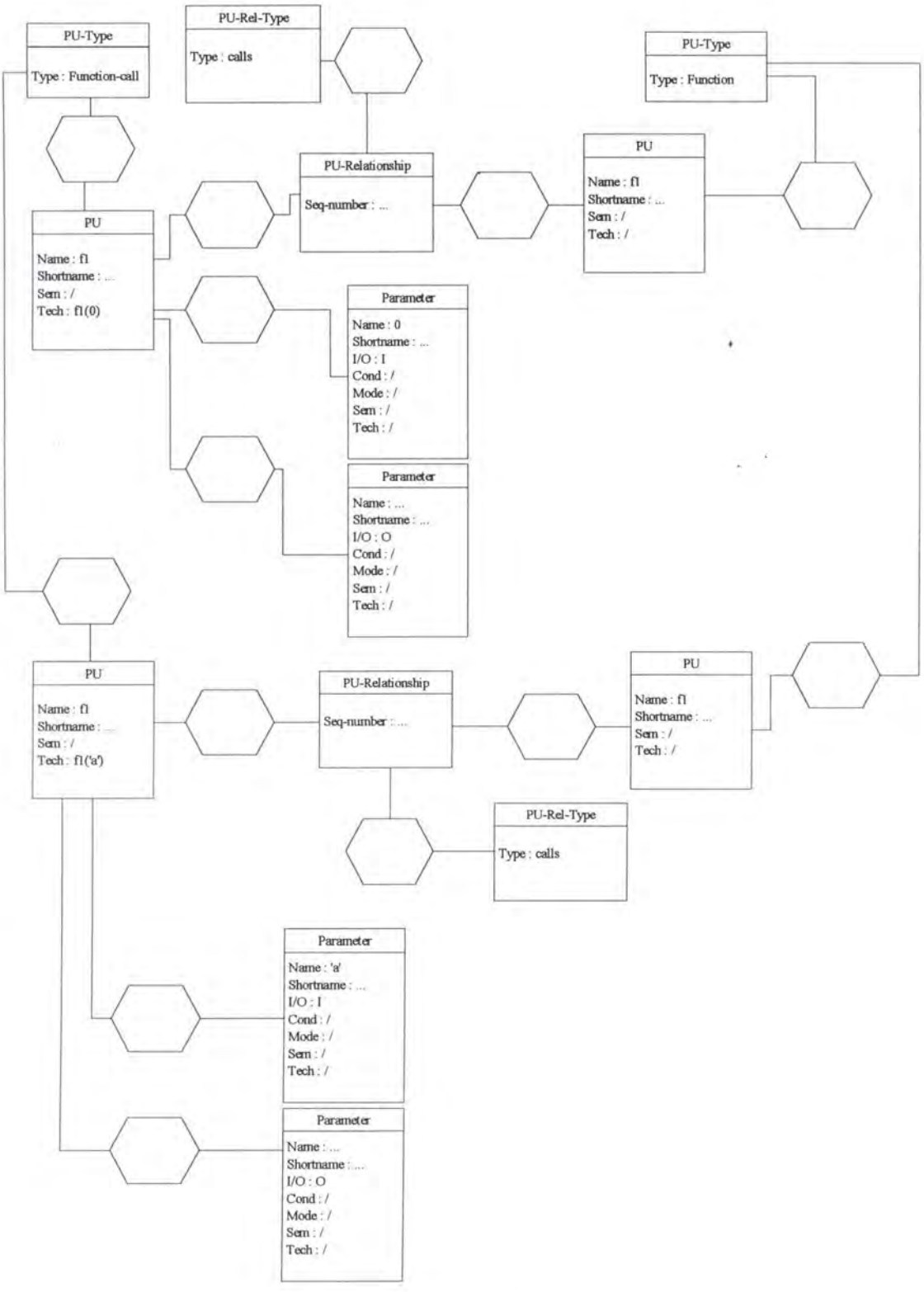


Figure 4-47 : C++ - Appel d'une fonction surchargée

2) Surcharge d'opérateurs

Exemple [ACH93]

```
class a
{
        public :
                friend a operator+(a,a);        // Version I
                friend a operator+(int, a);     // Version II
                friend a operator+(a, int);     // Version III
};
a operator+(a p1, a p2)
{
...
};
a operator+(int p1, a p2)
{
...
};
a operator+(a p1, int p2)
{
...
};
main()
{
        a a1, a2, a3
        a1 = a2 + a3;                   // Appel de la version I
        a1 = a2 + 10;                   // Appel de la version II
        a1 = 10 + a2;                   // Appel de la Version III
}
```

Exemple 4-16 : C++ - Définition d'un opérateur surchargé

Explication

Le mode de représentation d'un opérateur surchargé (voir Exemple 4-16) est le même que celui d'une fonction surchargée. Nous créons une *processing unit* par opérateur, et nous relions chacun d'eux à ses paramètres en entrée et en sortie (voir Figure 4-48).

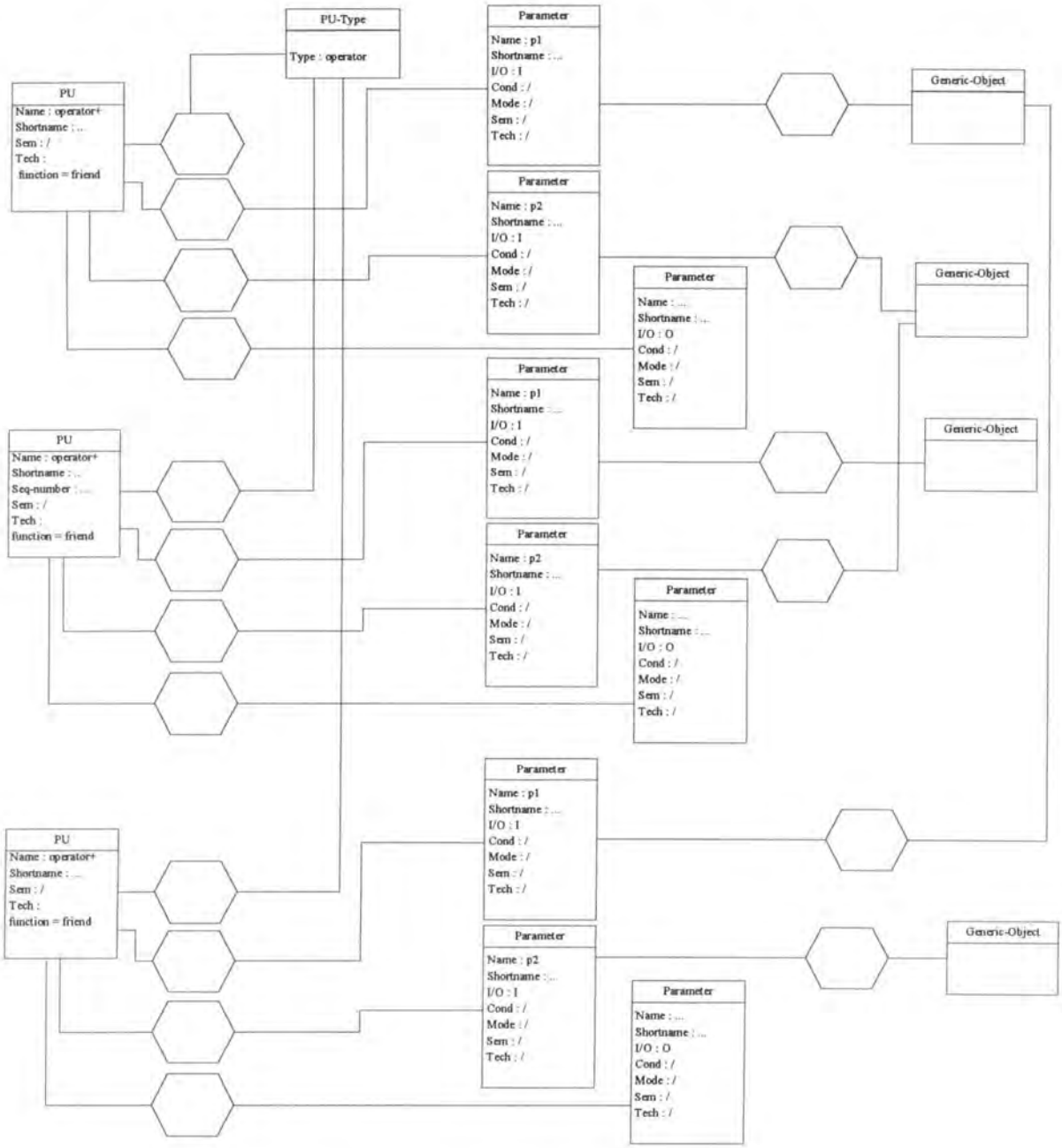


Figure 4-48 : C++ - Modélisation d'un opérateur surchargé

Lors de l'appel de l'opérateur la *processing unit* adéquat sera relié à la *processing unit* de type *function-call* (voir Figure 4-49).

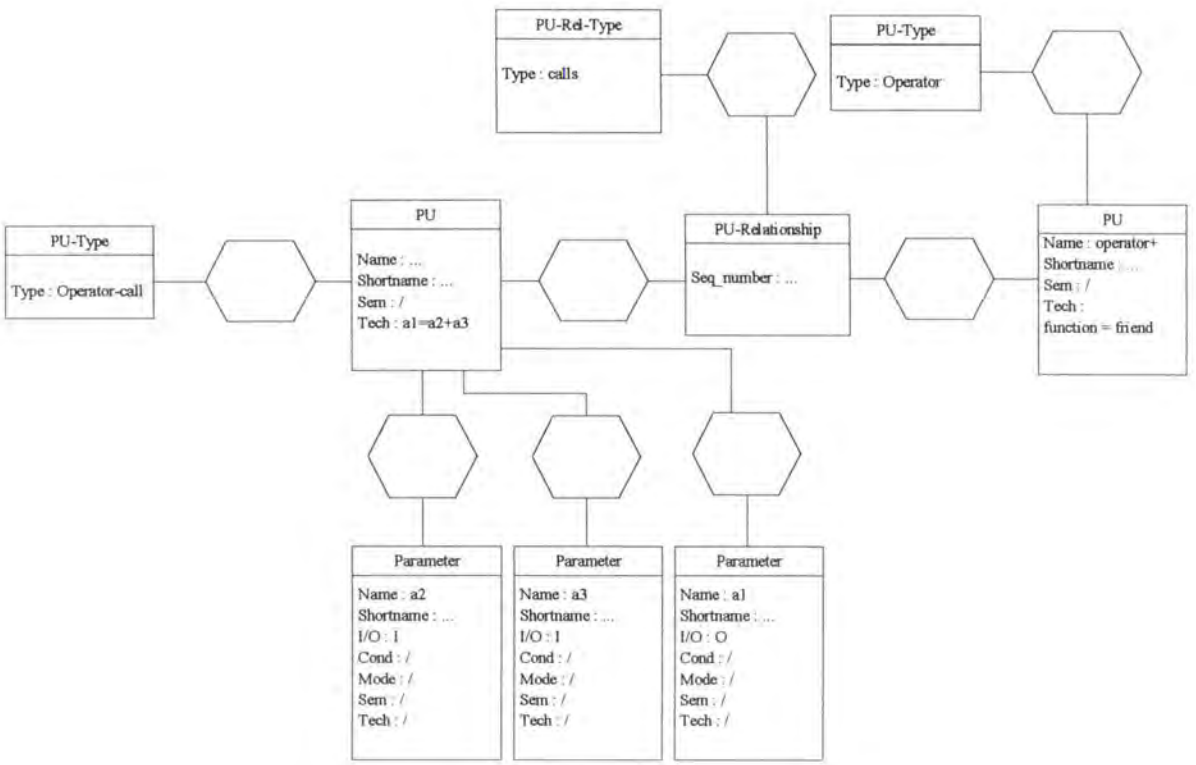


Figure 4-49 : C++ - Appel de l'opérateur surchargé

4.10 Pascal

Le langage Pascal étant très proche du langage C, nous n'allons pas refaire une énumération détaillée de toutes les structures possibles. La seule différence de ce langage par rapport au langage C est qu'il existe une structure supplémentaire, à savoir la boucle *repeat*.

La boucle repeat

Cette boucle est très semblable à la boucle do ... while du langage C. La différence est que, pour la boucle repeat, le type de la *processing unit* sera un nom particulier représentant le type de la boucle (voir Figure 4-50).

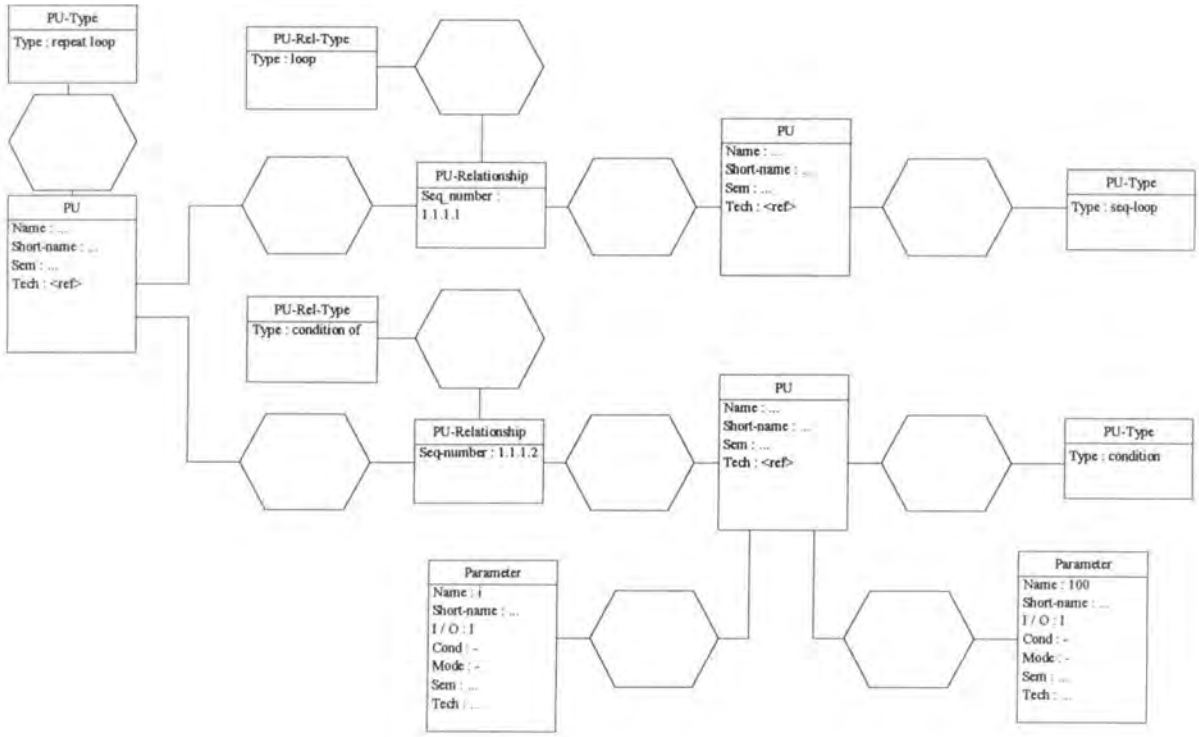


Figure 4-50 : Pascal - Modélisation d'une instruction repeat

4.11 PML

Considérons le programme PML suivant (voir Exemple 4-17) :

```
classes

  Rapport isa Entity with

    parts
      texte : String := ''

  end with

  Personne_1 isa Role with

    resources
      rapport : Rapport
      envoit : giveport Rapport
      envoye : Bool := false
      termine : Bool := false

    actions
      ecrire :
        GetNew ( agendaLabel = 'Redaction Rapport',
                 class = Rapport,
                 label = 'Rapport',
                 object = rapport )
      when
        (isnil ecrire)

      envoyer :
        {ViewObject  ( agendaLabel = 'Envoi',
                       icon = 'MailBox',
                       object = rapport );
         GiveCopy( gram = rapport,
                   interaction = envoit );
         envoye := true;
         termine := true}
      when
```

```
      (nonnil ecrire)

  termconds
    (termine = true)

end with

Personne_2 isa Role with

  resources
    rapport : Rapport
    recoit : takeport Rapport
    recu : Bool := false
    termine : Bool := false

  actions
    recevoir :
      {Take ( gram = rapport,
              interaction = recoit );
       ViewObject ( agendaLabel = 'Lire Rapport',
                    icon = 'MailBox',
                    object = rapport );
       recu := true;
       termine := true}
    when
      (recu = false)

  termconds
    (termine = true)

end with

resources
  rcpt : takeport Rapport
  emss : giveport Rapport
  var_P1 : Personne_1
  var_P2 : Personne_2

  actions
    createinteractions :
      {NewInteraction(giver = emss, taker = rcpt)}
    when
```

```
    (isnil createinteractions)

role_P1 :
  StartRole ( agendaLabel = 'Personne 1',
              envoit = emss,
              roleClass = Personne_1,
              roleInst = var_P1)
when
  (isnil role_P1 & nonnil createinteractions)

role_P2 :
  StartRole ( agendaLabel = 'Personne 2',
              roleClass = Personne_2,
              roleInst = var_P2,
              recoit = rcpt)
when
  (isnil role_P2 & nonnil createinteractions)
```

Exemple 4-17 : PML - Un programme

L'entité "rapport" (voir Exemple 4-17) sera représentée de la façon suivante (voir Figure 4-51) :

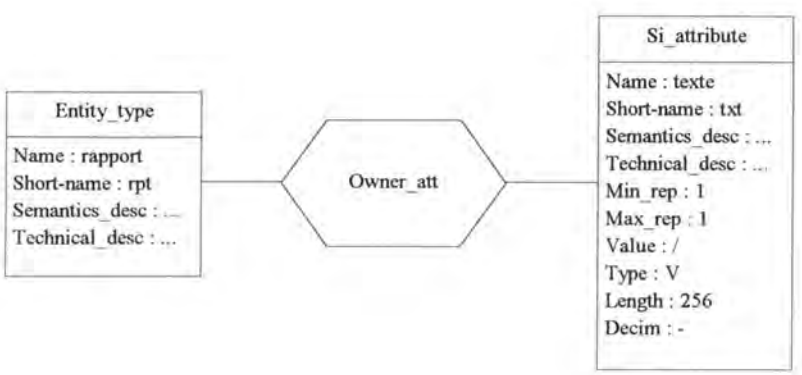


Figure 4-51 : PML - Modélisation d'une entité

Le rôle "Personne_1" sera également représenté par un type d'entité. Les variables locales (attribut *resources* du rôle) constitueront les attributs de ce type d'entité (voir Figure 4-52).

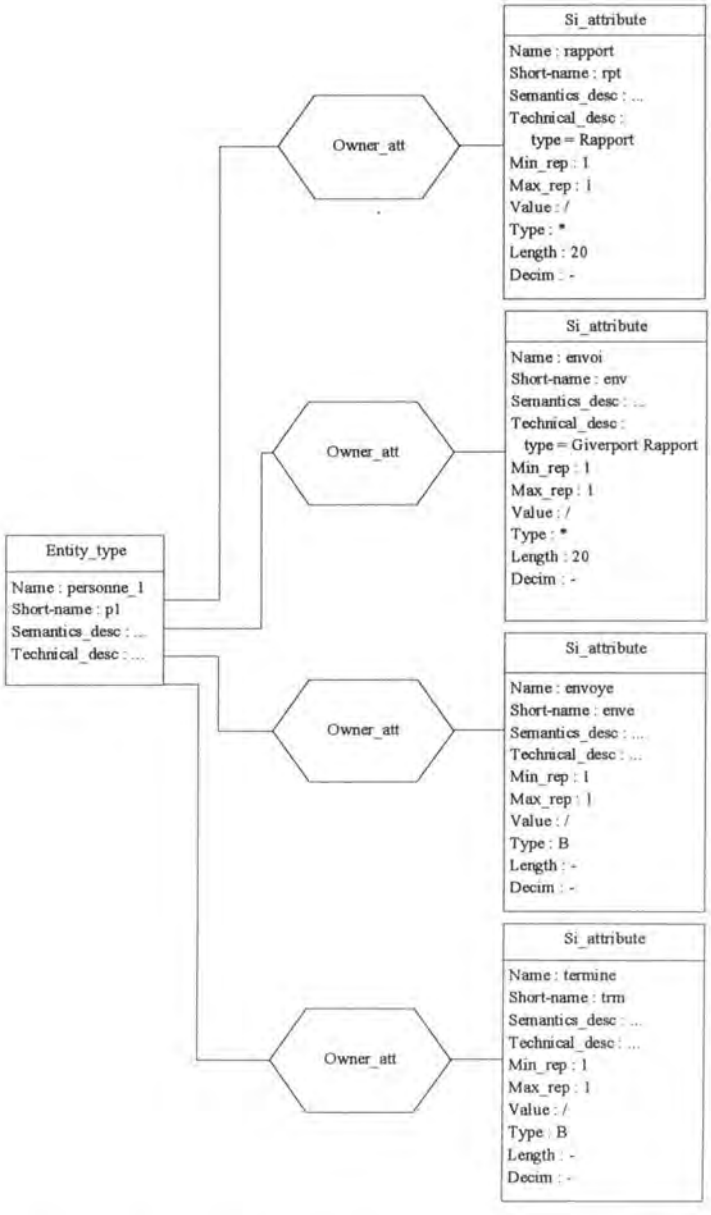


Figure 4-52 : PML - Modélisation de la partie donnée d'un rôle

Un rôle est un objet encapsulant, dans une même définition, des déclarations de variables et de traitements. Nous avons déjà dit qu'un rôle était représenté par une *entity type*. Les traitements, quant à eux, seront représentés par des *processing units*. Pour modéliser le lien entre les rôles et leurs traitements, nous utilisons le concept de *generic object*. Pour chaque objet représenté dans DB-Main il existe un *generic object*. Nous allons donc considérer que le *generic object* correspondant au rôle est un paramètre de la *processing unit* représentant ses traitements (voir Figure 4-53).

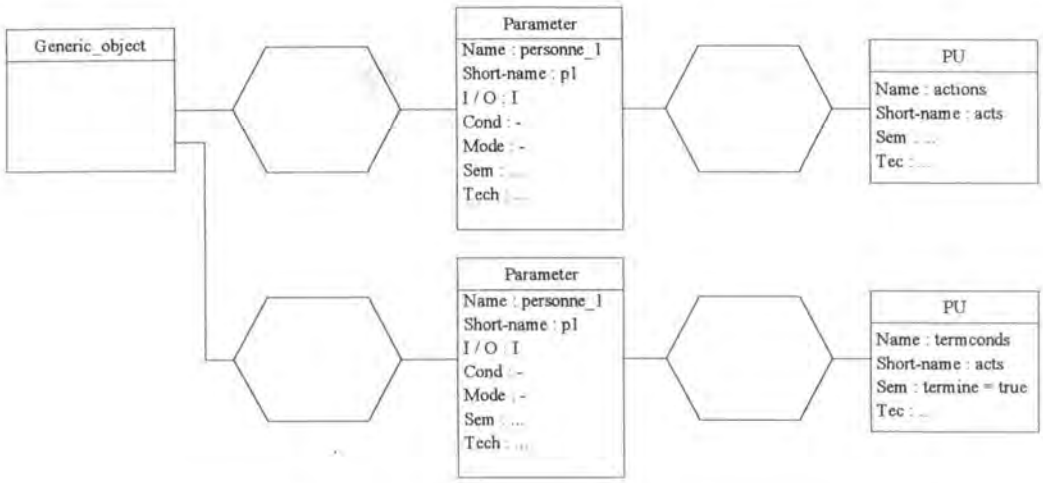


Figure 4-53 : PML - Modélisation de la partie traitements d'un rôle

En ce qui concerne les actions, les conditions de terminaison, les hypothèses initiales et les invariants, nous utiliserons le concept de *processing unit*. Les actions seront ensuite décomposées en instructions, chacune d'entre elles étant une *processing unit*. Elles seront reliées à l'action grâce à l'entité *PU-Relationship* définissant le type de relation entre *processing units* (voir Figure 4-54).

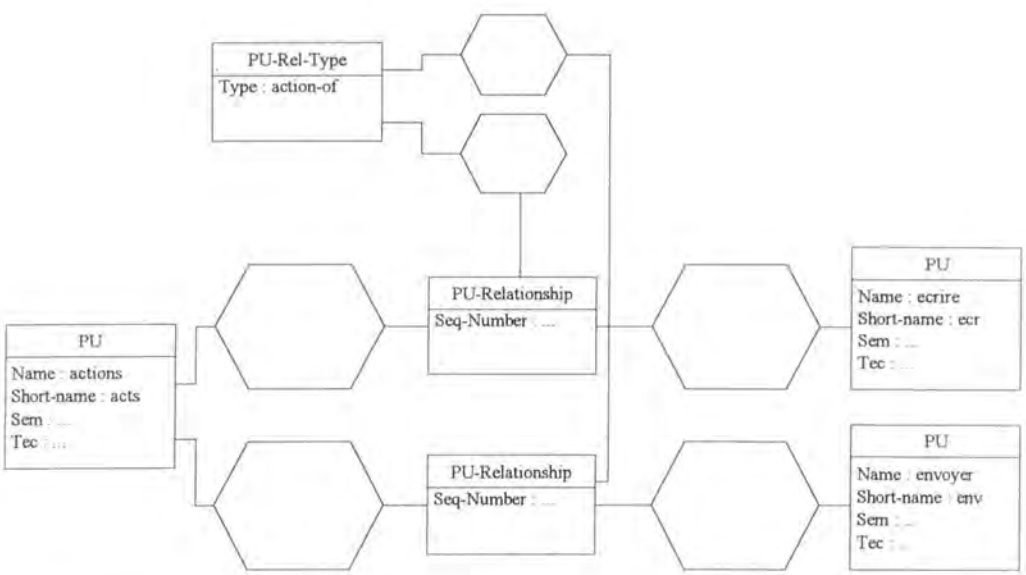


Figure 4-54 : PML - Décomposition de la partie traitements d'un rôle en actions

Les interactions entre rôles seront modélisées par des *Rel-Types*. Les variables PML représentant les deux extrémités du canal de communication seront, quant à elles, des attributs de l'*entity-type* représentant le rôle.

Il reste maintenant à représenter les agents PML. Pour cela, nous disposons d'une entité *actor* nous permettant de définir la personne ou le groupe de personnes à qui appartient un traitement ou une donnée (voir Figure 4-55).

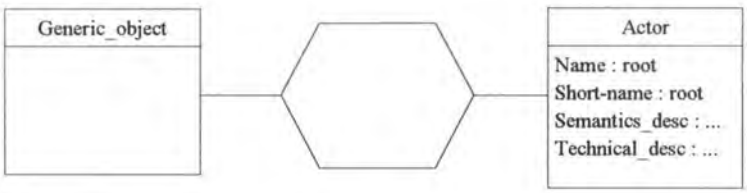


Figure 4-55 : PML - Affectation d'un rôle à un acteur

5. Analyse d'un exemple complet : Cobol

Cette partie va être consacrée à l'étude d'un programme Cobol complet (voir Exemple 5-1). Pour ce faire, nous allons commencer par donner le listing de ce programme avant de montrer comment nous pouvons le représenter. Nous ne développerons bien sûr pas le programme dans son entièreté, mais nous allons nous attaquer aux différentes structures que nous pouvons rencontrer.

Le programme ci-dessous est tiré de [NEW86].

```
IDENTIFICATION DIVISION.

PROGRAM-ID. PROGPAYE.

AUTHOR.LARRY NEWCOMER.
INSTALLATION.PENN STATE UNIVERSITY -- YORK CAMPUS.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-3081 WITH DEBUGGING MODE.
OBJECT-COMPUTER. IBM-3081.
```

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT FI-HEUR-VALID
                ASSIGN TO HEURDISK
                ORGANIZATION IS SEQUENTIAL
                ACCESS IS SEQUENTIAL
        .
        SELECT FI-EMPL-MAIT
                ASSIGN TO EMPLMAST
                ORGANIZATION IS SEQUENTIAL
                ACCESS IS SEQUENTIAL
        .
        SELECT FI-CHEQUES-PAYE
                ASSIGN TO CHEQUES
                ORGANIZATION IS SEQUENTIAL
                ACCESS IS SEQUENTIAL
        .
        SELECT FI-REG-PAYE
                ASSIGN TO CHEQUES
                ORGANIZATION IS SEQUENTIAL
                ACCESS IS SEQUENTIAL
        .
DATA DIVISION.

FILE SECTION.

FD      FI-HEUR-VALID
        BLOCK CONTAINS 0 RECORDS
        RECORD CONTAINS 80 CHARACTERS
        LABEL RECORDS ARE OMITTED
        .

01      ENR-VALHEUR.
        05      VAL-ID          PIC X(5).
        05      VAL-HEURES      PIC S9(2)V9.
        05      VAL-DATFERM     PIC X(6).
        05      VAL-DEPART      PIC X(4).
        05      VAL-DATED       PIC X(6).
        05      FILLER          PIC X(56).

FD      FI-EMPL-MAIT
        BLOCK CONTAINS 0 RECORDS
        RECORD CONTAINS 80 CHARACTERS
        LABEL RECORDS ARE STANDARD
        .

01      ENR-EMPL-MAIT.
        05      MAIT-ID             PIC X(5).
        05      MAIT-NOM            PIC X(20).
        05      MAIT-TAUX           PIC S9(3)V99.
        05      MAIT-BRUT-A-DATE    PIC 9(7)V99.
        05      MAIT-TAXE-A-DATE    PIC 9(6)V99.
        05      MAIT-NET-A-DATE     PIC 9(7)V99.
        05      FILLER              PIC X(24).

FD      FI-CHEQUES-PAYE
        BLOCK CONTAINS 0 RECORDS
        RECORD CONTAINS 66 CHARACTERS
        LABEL RECORDS ARE STANDARD
        .
```

```
01      ENR-CHEQUES.
        05      CHQ-ID          PIC X(5).
        05      CHQ-HEUR        PIC S9(2)V9.
        05      CHQ-DATE        PIC X(6).
        05      CHQ-DEPART      PIC X(4).
        05      CHQ-TAUX        PIC S9(3)V99.
        05      CHQ-NOM         PIC X(20).
        05      CHQ-BRUT        PIC S9(6)V99.
        05      CHQ-IMPOT       PIC S9(5)V99.
        05      CHQ-NET         PIC S9(6)V99.

FD      FI-REG-PAYE
        RECORD CONTAINS 132 CHARACTERS
        LABEL RECORDS ARE OMITTED

01      LIGNE-PAIE              PIC X(132).

WORKING-STORAGE SECTION.

01      AIGUIL-PROG.
        05      WS-AIG-FIN-HEUR         PIC X(3).
                88      PLUS-HEUR       VALUE "OUI".
                88      HEUR-DISPONI    VALUE "NON".
        05      WS-FIN-MAIT             PIC X(3).
                88      PLUS-MAIT       VALUE "OUI".
                88      MAIT-DISPONI    VALUE "NON".

01      COMPT-PROG.
        05      WS-NOPAGE               PIC S9(3)       COMP-3.
        05      WS-NBRE-EMPL            PIC S9(5)       COMP-3.
        05      WS-NB-A-SAUT            PIC S9          COMP SYNC.
        05      WS-LIG-PAGE             PIC S9(2)       COMP SYNC.


01      DONNEES-IMPOTS.
        05      VALEURS-IMPOTS.
                10      FILLER          PIC S9(3)V99    COMP-3 VALUE +800.00.
                10      FILLER          PIC SV999       COMP-3 VALUE  +.25.
                10      FILLER          PIC S9(3)V99    COMP-3 VALUE  +500.00.
                10      FILLER          PICSV999        COMP-3 VALUE  +.15.
                10      FILLER          PIC S9(3)V99    COMP-3 VALUE  +300.00.
                10      FILLER          PIC SV999       COMP-3 VALUE  +.08.
                10      FILLER          PIC S9(3)V99    COMP-3 VALUE ZERO.
                10      FILLER          PIC SV999       COMP-3 VALUE  +.05.
        05      TABLE-IMPOTS  REDEFINES VALEURS-IMPOT
                                        OCCURS 4 TIMES
                                        INDEXED BY IND-TAB.
                10      LIM-BASSE       PIC S9(3)V99    COMP-3.
                10      TAUX-IMP        PIC SV999       COMP-3.

01      ZONES-TOT-PROG.
        05      WS-HEUR-TOT             PIC S9(7)V9     COMP-3.

01      ZONES-CALCUL-PROG.
        05      WS-BRUT                 PIC S9(7)V99    COMP-3.
        05      WS-NET                  PIC S9(7)V99    COMP-3.
        05      WS-IMPPOT               PIC S9(5)V99    COMP-3.

01      CONST-PROG
        05      WS-TAILLE-PGE  PIC S9(2)        COMP SYNC.
                                        VALUE +50.
        05      WS-SAUT-AV-E-T          PIC S9(2)       COMP SYNC.
```

```
                                    VALUE +2.
    05      WS-SAUT-AV-P-P          PIC S9(2)       COMP SYNC.
                                    VALUE +4.
    05      WS-SAUT-AV-DET          PIC S9(2)       COMP SYNC.
                                    VALUE +2.
    05      WS-MESSAGE-NON-APPAREIL                 PIC X(34)
                                    VALUE "L'EMPLOYE N'EST PAS SUR LE FICHIER".
    05      WS-MAITRE-NON-APPAREIL                  PIC X(28)
                                    VALUE "INACTIF DURANT CETTE PERIODE".
    05      WS-POINT-HSUPP          PIC S99V9       COMP-3 VALUE +40.0.
    05      WS-FACT-HSUPP PIC S9V9          COMP-3 VALUE +1.5.

01  WS-DATE-SYS.
    05      SYS-AA          PIC 99.
    05      SYS-MM          PIC 99.
    05      SYS-JJ          PIC 99.

01  WS-E-T-1.
    05      FILLER                  PIC X(2) VALUE SPACES.
    05      FILLER                  PIC X(16)       VALUE "REGISTRE DE PAYE".
    05      WS-ET-JJ                PIC Z9.
    05      FILLER                  PIC X           VALUE "/".
    05      WS-ET-MM                PIC 99.
    05      FILLER                  PIC X           VALUE "/".
    05      WS-ET-AA                PIC 99.
    05      FILLER                  PIC X(3) VALUE SPACES.
    05      FILLER                  PIC X(5) VALUE "PAGE".
    05      WS-ET-PAGE              PIC ZZ9.
    05      FILLER                  PIC X(93)       VALUE SPACES.

01  WS-E-T-2.
    05      FILLER                  PIC X(7) VALUE "IDENT".
    05      FILLER                  PIC X(6) VALUE "HEURES".
    05      FILLER                  PIC X(8) VALUE "DATE".
    05      FILLER                  PIC X(8) VALUE "DEPART".
    05      FILLER                  PIC X(6) VALUE "TAUX".
    05      FILLER                  PIC X(97)       VALUE SPACES.

01  WS-LIGNE-DETAIL.
    05 WS-DET-ID                    PIC X(5).
    05      FILLER                  PIC X(1) VALUE SPACES.
    05      WS-DET-HEUR             PIC ZZ9.
    05      FILLER                  PIC X(2) VALUE SPACES.
    05      WS-DET-DATE             PIC X(6).
    05      FILLER                  PIC X(2) VALUE SPACES.
    05      WS-DET-DEPART           PIC X(4).
    05      FILLER                  PIC X(2) VALUE SPACES.
    05      WS-DET-TAUX             PIC ZZZ.99.
    05      FILLER                  PIC X(2) VALUE SPACES.
    05      WS-REM-DET.
            10      WS-DET-PAYE     PIC ZZZ.ZZZ.99.
            10      FILLER          PIC X(88).

01  WS-LIGNE-PP.
    05      FILLER                  PIC X(21)       VALUE "NOMBRE D'EMPLOYES = ".
    05      WS-PP-CPTE              PIC ZZ,ZZ9.
    05      FILLER                  PIC X(3)        VALUE SPACES.
    05      FILLER                  PIC X(16)       VALUE "TOTAL HEURES = ".
    05      WS-PP-HEUR              PIC Z,ZZZ,ZZZ.9.
    05      FILLER                  PIC X(75)       VALUE SPACES.
    05      WS-ZONE-SORTIE          PIC X(132).
```

```
PROCEDURE DIVISION.

CREER-FICH-CHEQUES-PAYE.

        PERFORM 100-INITIALISER
        PERFORM 200-APPAREILLER-HEURES-Et-MAITRE.
                UNTIL PLUS-HEUR AND PLUS-MAIT
        PERFORM 300-TERMINER
        STOP RUN

100-INITIALISER.

        OPEN    INPUT           FI-HEUR-VALID
                I-O             FI-EMPL-MAIT
                OUTPUT          FI-CHEQUES-PAYE
                                FI-REG-PAYE
        MOVE "NON" TO           WS-AIG-FIN-HEUR
                                        WS-FIN-MAIT
        MOVE ZERO TO            WS-NOPAGE
                                        WS-NBRE-EMPL
                                        WS-HEUR-TOT
        MOVE WS-TAILLE-PGE TO           WS-LIG-PAGE
        ACCEPT WS-DATE-SYSTEME FROM DATE
        MOVE SYS-AA TO                  WS-ET-AA
        MOVE SYS-MM TO                  WS-ET-MM
        MOVE SYS-JJ TO                  WS-ET-JJ

        PERFORM 270-ENTRER-HEURES
        PERFORM 280-ENTRER-MAITRE

200-APPAREILLER-HEURE-ET-MAITRE.

        MOVE SPACES TO WS-LIGNE-DETAIL
        IF VAL-IF EQUAL MAIT-ID
                PERFORM 210-CREER-ENREG-PAYE
                PERFORM 215-MAJ-FICH-MAITRE
                PERFROM 270-ENTRER-HEURES
                PERFORM 280-ENTRER-MAITRE
        ELSE IF VAL-ID LESS THAN MAIT-ID
                MOVE WS-MAITRE-NON APPAREIL     TO      WS-REM-DET
                MOVE VAL-ID                     TO      WS-DET-ID
                MOVE VAL-HEURES                 TO      WS-DET-HEUR
                MOVE VAL-DATFERM                TO      WS-DET-DAT
                MOVE VAL-DEPART                 TO      WS-DET-DEPART
                MOVE WS-SAUT-AV-DET             TO      WS-NB-A-SAUT
                MOVE WS-LIGNE-DETAIL            TO      WS-ZONE-SORTIE
                        PERFORM 270-ENTRER-HEURES
            ELSE
                MOVE WS-MESSAGE-NON-APPAREIL    TO      WS-REM-DET
                MOVE MAIT-ID                    TO      WS-DET-ID
                MOVE MAIT-TAUX                  TO      WS-DET-TAUX
                MOVE WS-SAUT-AV-DET             TO      WS-NB-A-SAUT
                MOVE WS-LIGNE-DETAIL            TO      WS-ZONE-SORTIE
                PERFORM 280-ENTRER-MAITRE

        PERFORM 260-IMPR-REGISTRE


210-CREER-ENREG-PAYE.

        MOVE VAL_ID             TO      WS-DET-ID
        MOVE VAL-HEURES         TO      WS-DET-HEUR
        MOVE VAL-DATFERM        TO      WS-DET-DAT
```

```
        MOVE VAL-DEPART          TO    WS-DET-DEPART
        MOVE MAIT-TAUX           TO    WS-DET-TAUX
        MOVE WS-SAUT-AV-DET      TO    WS-NB-A-SAUT

        PERFORM 220-CALCUL-PAYE
        MOVE WS-NET              TO    WS-DET-PAYE
        PERFORM 250-ECR-ENR-PAYE
        MOVE WS-LIGNE-DETAIL     TO    WS-ZONE-SORTIE
        ADD 1                    TO    WS-NBRE-EMPL
        ADD VAL-HEURES           TO    WS-HEUR-TOT
        .

220-CALCUL-PAYE.

        PERFORM 230-CALCUL-BRUT
        PERFORM 240-CALCUL-IMPOT
        SUBSTRACT WS-IMPOT FROM WS-BRUT GIVING WS-NET
        .

215-MAJ-FICH-MAITRE.

        ADD WS-IMPOT TO MAIT-TAX-A-DATE
        ADD WS-BRUT TO MAIT-BRUT-A-DATE
        ADD WS-NET TO MAIT-NET-A-DATE
        REWRITE ENR-EMPL-MAIT
        .

230-CALCUL-BRUT.

        IF VAL-HEURES GREATER THAN WS-POINT-HSUPP
                COMPUTE WS-BRUT     =     (WS-POINT-HSUPP * MAIT-TAUX)
                                          + ((VAL-HEURES  -  WS-POINT-HSUPP)
                                                  * MAIT-TAUX * WS-FACT-H-SUPP)
        ELSE
                COMPUTE WS-BRUT = VAL-HEURES * MAIT-TAUX
        .

240-CALCUL-IMPOT.

        SET IND-TAB TO 1
        SEARCH TABLE-IMPOTS
                WHEN WS-BRUT GREATER THAN LIM-BASSE
                        COMPUTE WS-IMPOT = WS-BRUT * TAUX-IMP(IND-TAB)
        .

250-ECR-ENREG.PAYE.

        MOVE VAL-ID              TO    CHQ-ID
        MOVE VAL-HEURES          TO    CHQ-HEUR
        MOVE VAL-DATFERM         TO    CHQ-DATE
        MOVE VAL-DEPART          TO    CHQ-DEPART
        MOVE MAIT-TAUX           TO    CHQ-TAUX
        MOVE MAIT-NOM            TO    CHQ-NOM
        MOVE WS-BRUT             TO    CHQ-BRUT
        MOVE WS-IMPOT            TO    CHQ-IMPOT
        MOVE WS-NET              TO    CHQ-NET
        WRITE ENR-CHEQUES
        .

260-IMPR-REGISTRE.

        IF WS-LIG-PAGE + WS-NB-A-SAUT
                GREATER THAN WS-TAILLE-PAGE
```

```
                ADD 1 TO WS-NOPAGE
                MOVE WS-NOPAGE TO WS-ET-PAGE
                WRITE LIGNE-PAIE
                        FROM WS-ET-1
                        AFTER ADVANCING PAGE
                        WRITE LIGNE-PAIE
                                FROM WS-ET-2
                                AFTER ADVANCING WS-SAUT-AV-E-T
                        MOVE WS-SAUT-AV-E-T TO WS-LIG-PAGE

                WRITE LIGNE-PAIE
                        FROM WS-ZONE-SORTIE
                        AFTER ADVANCING WS-NB-A-SAUT LINES
                ADD WS-NB-A-SAUT TO WS-LIG-PAGE

270-ENTRER-HEURES.

        READ FI-HEUR-VALID
                AT END
                        MOVE "OUI" TO WS-FIN-HEUR
                        MOVE HIGH-VALUES TO VAL-ID

280-ENTRER-MAITRE.

        READ FI-EMPL-MAIT
                AT END
                        MOVE "OUI" TO WS-FIN-MAIT
                        MOVE HIGH-VALUES TO MAIT-ID

300-TERMINER.

        MOVE WS-NBRE-EMPL       TO      WS-PP-CPTE
        MOVE WS-HEUR-TOT        TO      WS-PP-HEUR
        MOVE WS-LGNE-PP         TO      WS-ZONE-SORTIE
        MOVE WS-SAUT-AV-P-P     TO      WS-NB-A-SAUT
        MOVE WS-LIGNE-PP        TO      WS-ZONE-SORTIE
        PERFORM 260-IMPR-REGISTRE

        CLOSE   FI-HEUR-VALID
                FI-EMPL-MAIT
                FI-CHEQUES-PAYE
                FI-REG-PAYE
```

Exemple 5-1 : Programme Cobol

Nous allons débuter cette analyse par montrer comment nous pouvons représenter les instructions de création d'un fichier. Pour ce faire, nous avons choisi l'exemple du fichier *fi-heur-valid* (voir Exemple 5-2).

```
SELECT FI-HEUR-VALID
        ASSIGN TO HEURDISK
        ORGANIZATION IS SEQUENTIAL
        ACCESS IS SEQUENTIAL
...

FD      FI-HEUR-VALID
        BLOCK CONTAINS 0 RECORDS
        RECORD CONTAINS 80 CHARACTERS
        LABEL RECORDS ARE OMITTED


01      ENR-VALHEUR.
        05      VAL-ID          PIC X(5).
        05      VAL-HEURES      PIC S9(2)V9.
        05      VAL-DATFERM     PIC X(6).
        05      VAL-DEPART      PIC X(4).
        05      VAL-DATED       PIC X(6).
        05      FILLER          PIC X(56).
```

Exemple 5-2 : Cobol - Création d'un fichier

Etant donné qu'il s'agit de la création d'un fichier, nous allons travailler sur le schéma des données persistantes. Nous allons donc y créer une *entity-type* qui est composée d'un seul attribut composé (*enr-valheur*). Cet attribut va donc être redécomposé pour correspondre à la structure ci-dessus (voir Exemple 5-2).

Les informations relatives à l'organisation du fichier vont être stockées dans les parties *technical-desc* des entités *entity-type* et *co-attribute* (voir Figure 5-1).

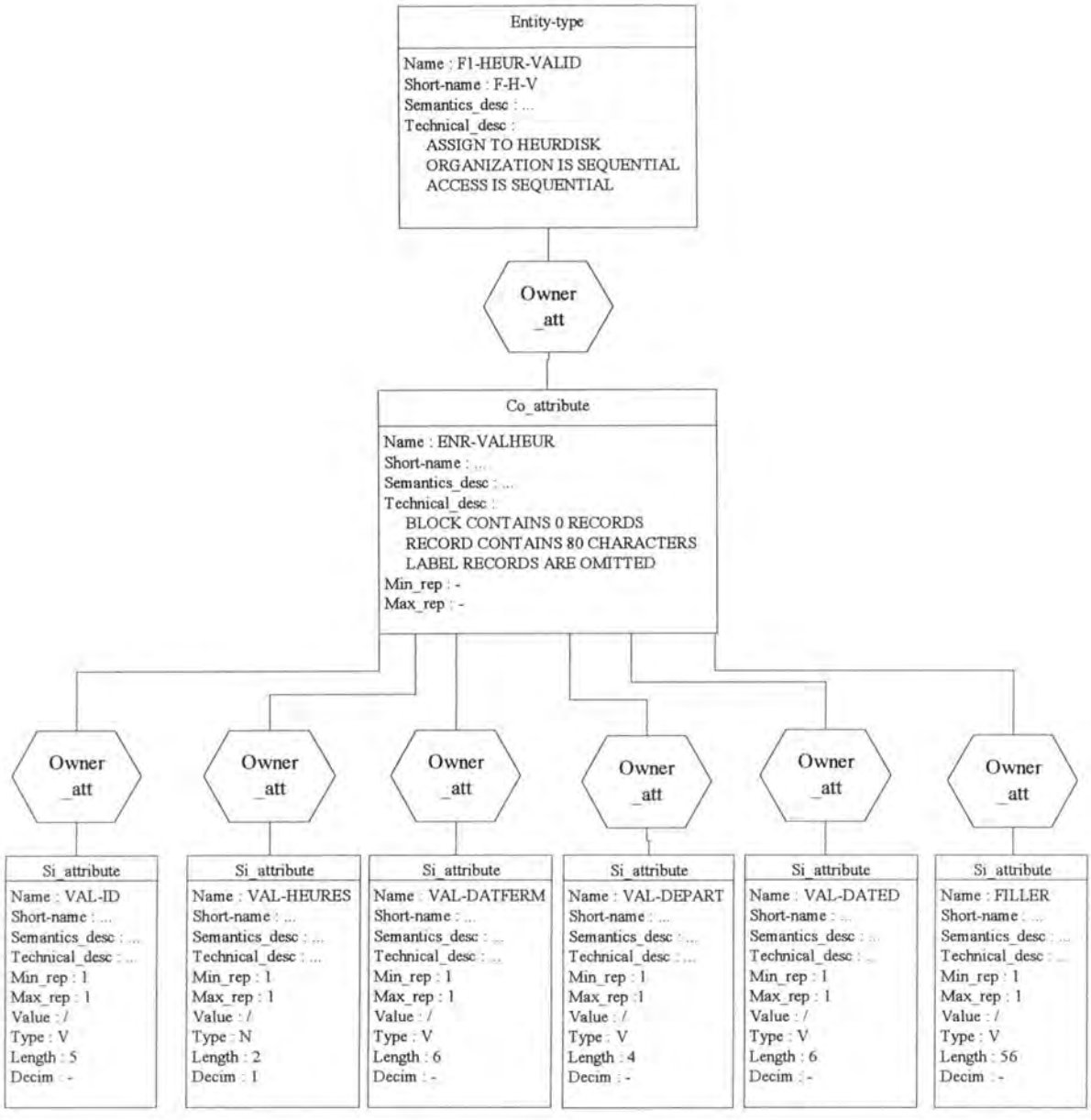


Figure 5-1 : Cobol - Modélisation de la définition d'un fichier

La définition d'une variable composée va se faire d'une manière similaire, à la différence près que l'*entity-type* créée appartient au schéma des données non-persistantes. Pour illustrer ce mécanisme, nous avons choisi l'exemple de la variable *aiguil-prog* (voir Exemple 5-3).

```
01      AIGUIL-PROG.
        05      WS-AIG-FIN-HEUR         PIC X(3).
                88      PLUS-HEUR       VALUE "OUI".
                88      HEUR-DISPONI    VALUE "NON".
```

```
05      WS-FIN-MAIT             PIC X(3).
        88      PLUS-MAIT       VALUE "OUI".
        88      MAIT-DISPONI    VALUE "NON".
```

Exemple 5-3 : Cobol - Définition d'une variable composée

Nous créons donc une *entity-type* qui est composée de deux attributs composés. Ces attributs sont eux aussi décomposés en deux attributs simples. Nous devons cependant spécifier que les attributs composés *ws-aig-fin-heur* et *ws-fin-mait* sont de type *PIC X(3)*. Nous sommes donc obligés de sauver cette information dans la partie *technical_desc* des entités *co_attribute* (voir Figure 5-2).

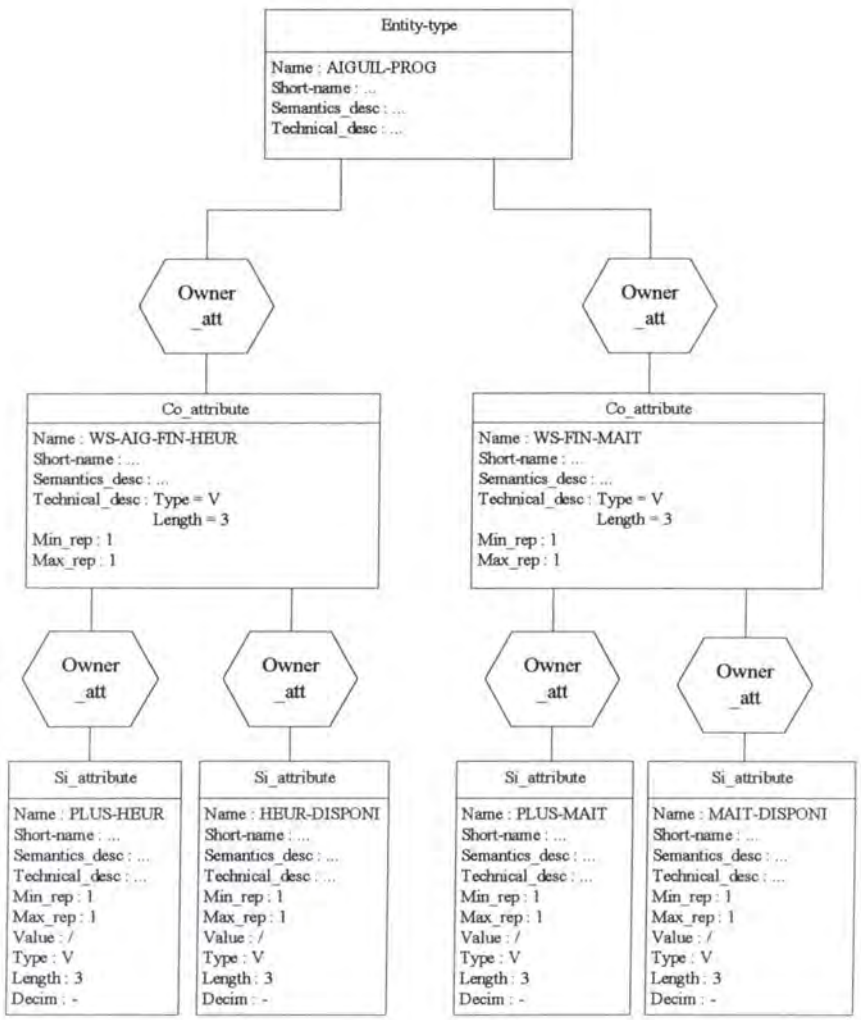


Figure 5-2 : Cobol - Modélisation de la définition d'une variable composée

L'exemple suivant (voir Exemple 5-4) est presque identique au précédent, mais il va nous permettre d'illustrer en plus les clauses *value*, *comp sync* et *comp-3*. Les deux dernières clauses sont, en fait, des informations relatives à des optimisations qui seront effectuées lors de la compilation.

```
01      CONST-PROG
        05      WS-TAILLE-PGE PIC S9(2)         COMP SYNC.
                                VALUE +50.
        05      WS-SAUT-AV-E-T  PIC S9(2)       COMP SYNC.
                                VALUE +2.
        05      WS-SAUT-AV-P-P  PIC S9(2)       COMP SYNC.
                                VALUE +4.
        05      WS-SAUT-AV-DET  PIC S9(2)       COMP SYNC.
                                VALUE +2.
        05      WS-MESSAGE-NON-APPAREIL         PIC X(34)
                                VALUE "L'EMPLOYE N'EST PAS SUR LE FICHIER".
        05      WS-MAITRE-NON-APPAREIL          PIC X(28)
                                VALUE "INACTIF DURANT CETTE PERIODE".
        05      WS-POINT-HSUPP  PIC S99V9       COMP-3 VALUE +40.0.
        05      WS-FACT-HSUPP   PIC S9V9        COMP-3 VALUE +1.5.
```

Exemple 5-4 : Cobol - Définition d'une variable composée

Nous allons utiliser l'attribut *value* pour spécifier la valeur que doit prendre l'attribut, tandis que l'attribut *technical_desc* contiendra l'information relative aux options de compilation (voir Figure 5-3).

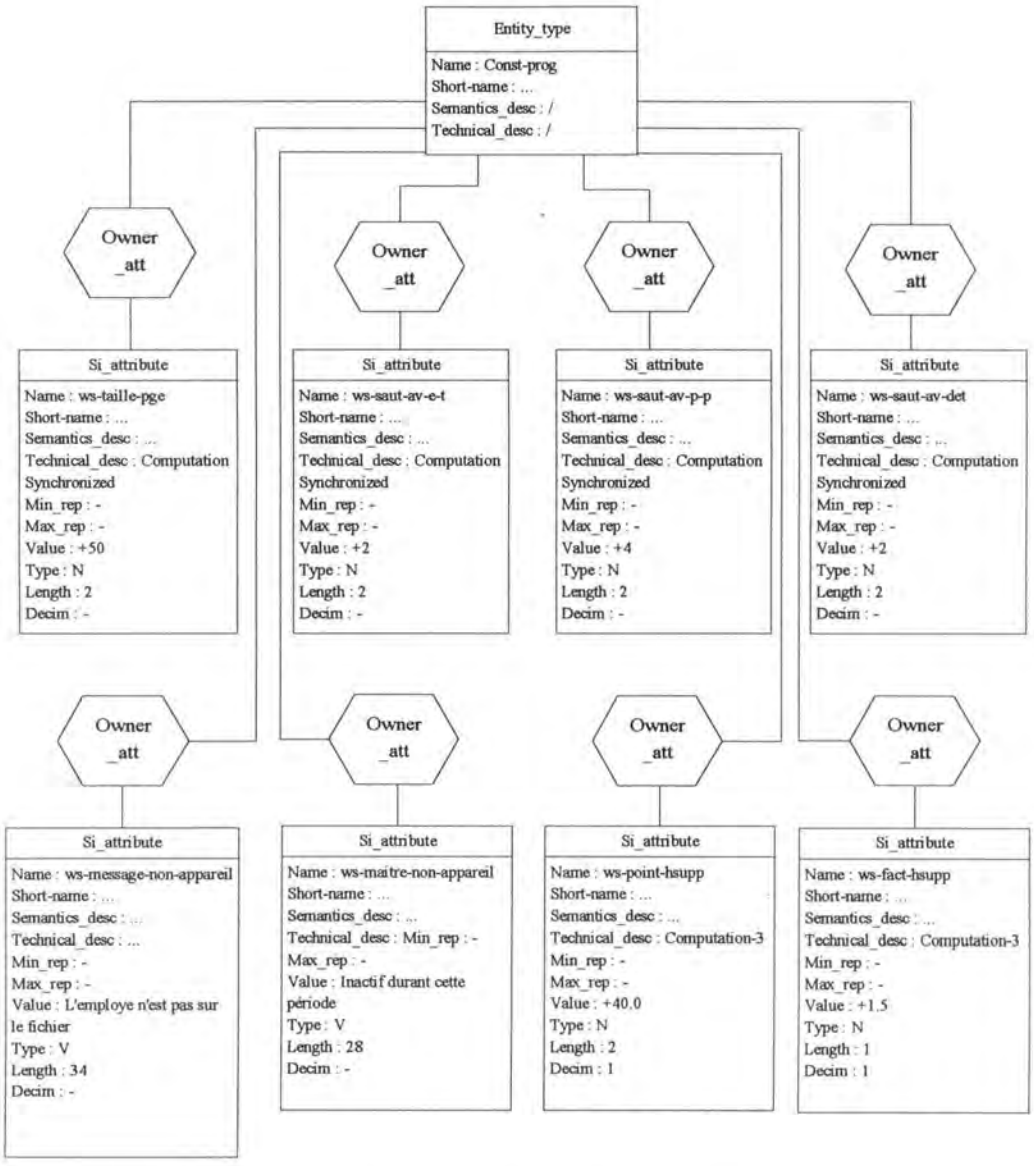


Figure 5-3 : Cobol - Modélisation de la définition d'une variable composée

Nous allons maintenant passer à l'analyse de la partie *procedure division* de notre programme.

Voyons comment nous pouvons représenter le programme principal (voir Exemple 5-5).

```
CREER-FICH-CHEQUES-PAYE.

        PERFORM 100-INITIALISER
        PERFORM 200-APPAREILLER-HEURES-Et-MAITRE.
                UNTIL PLUS-HEUR AND PLUS-MAIT
```

```
PERFORM 300-TERMINER
STOP RUN
```

Exemple 5-5 : Cobol - Programme principal

Nous créons tout d'abord une *processing unit* de type *main program* contenant une référence vers l'ensemble du programme principal. Cette entité possède des relations vers quatre autres *processing units*, chacune représentant une instruction du programme. Les trois premières constituent des appels de procédure, tandis que la dernière est l'instruction qui met fin à l'exécution du programme (voir Figure 5-4).

Chacune de ces procédures doit ensuite être représentée et un lien doit alors être créé entre l'appel et la définition des procédures. Ce lien sera matérialisé par une entité *PU-relationship* (voir Figure 5-5).

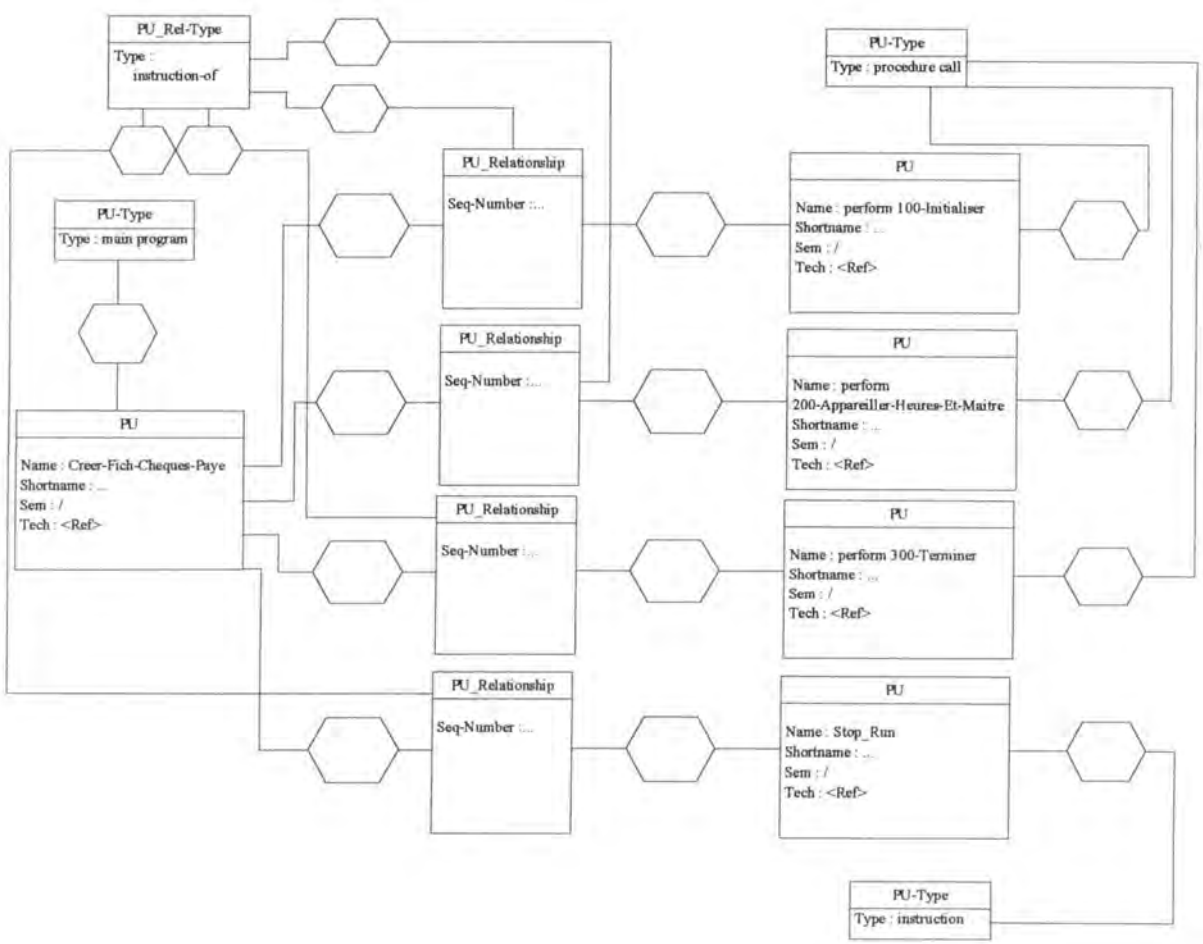


Figure 5-4 : Cobol - Modélisation du programme principal

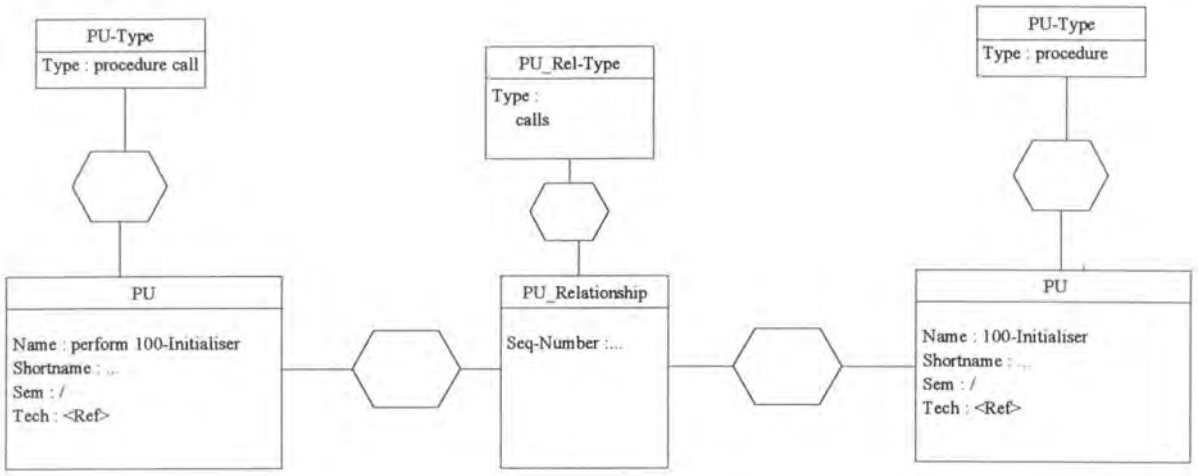


Figure 5-5 : Cobol - Lien entre une procédure et son appel

Nous allons maintenant voir comment nous pouvons représenter une procédure. Nous avons, pour cela, choisi l'exemple de la procédure *100-Initialiser* (voir Exemple 5-6).

```
100-INITIALISER.

        OPEN    INPUT           FI-HEUR-VALID
                I-O             FI-EMPL-MAIT
                OUTPUT          FI-CHEQUES-PAYE
                                FI-REG-PAYE
        MOVE "NON" TO           WS-AIG-FIN-HEUR
                                        WS-FIN-MAIT
        MOVE ZERO TO            WS-NOPAGE
                                        WS-NBRE-EMPL
                                        WS-HEUR-TOT
        MOVE WS-TAILLE-PGE TO           WS-LIG-PAGE
        ACCEPT WS-DATE-SYSTEME FROM DATE
        MOVE SYS-AA TO                  WS-ET-AA
        MOVE SYS-MM TO                  WS-ET-MM
        MOVE SYS-JJ TO                  WS-ET-JJ

        PERFORM 270-ENTRER-HEURES
        PERFORM 280-ENTRER-MAITRE
```

Exemple 5-6 : Cobol - Définition d'une procédure

La première étape consiste à relier l'entité contenant la procédure à ses différentes instructions, représentées chacune par une *processing unit*. Cela se fait, à nouveau, au moyen d'entités *PU-Relationship*. Les *processing units,* contenant les instructions, se voient attribuer un type indiquant la nature de l'instruction. (voir Figure 5-6).

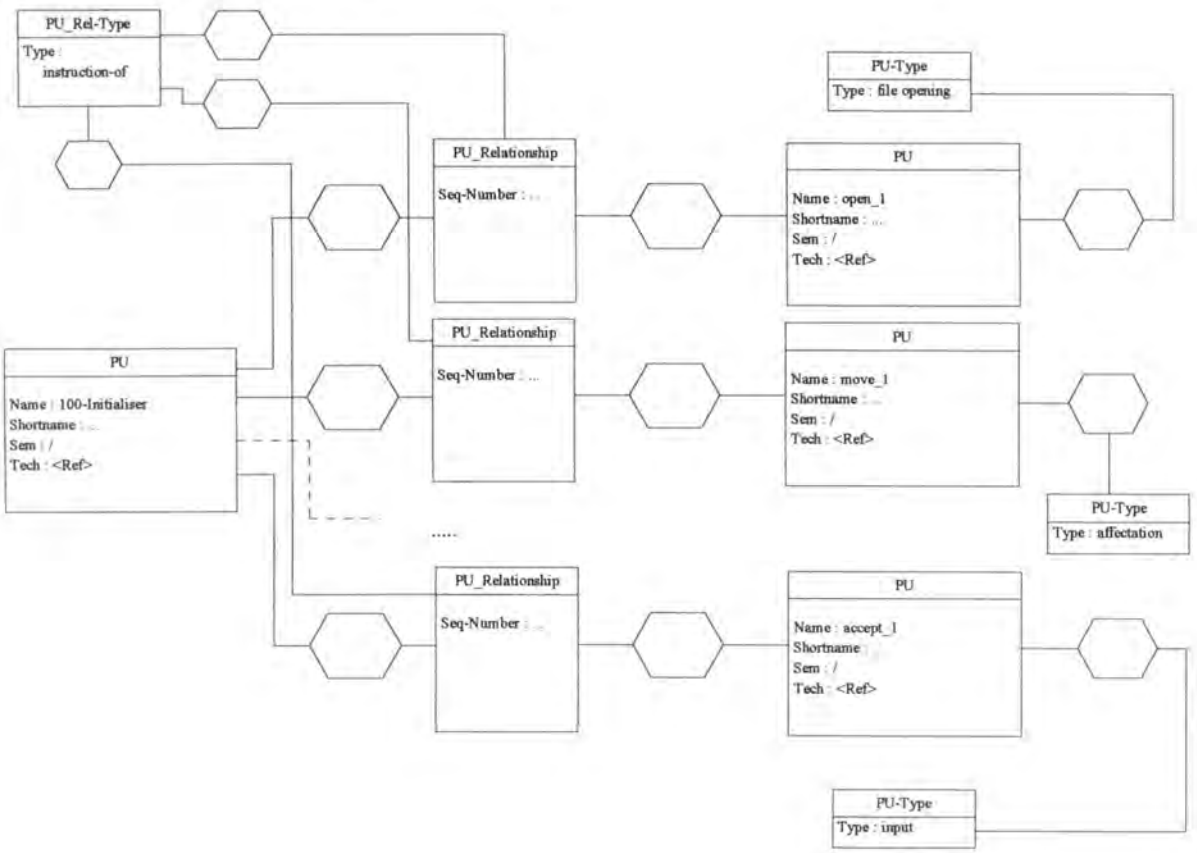


Figure 5-6 : Cobol - Modélisation d'une procédure

Chacune de ces *processing units* de type instruction est à nouveau décomposée. L'instruction, consistant à ouvrir les différents fichiers, est reliée à autant d'entités *parameter* qu'il n'y a de fichiers. Ces entités devront par la suite être reliées à leur *entity-type* correspondante. Le mode d'ouverture des fichiers (input, output ou input / output) est stocké dans l'attribut *I / O* de l'entité *parameter* (voir Figure 5-7).

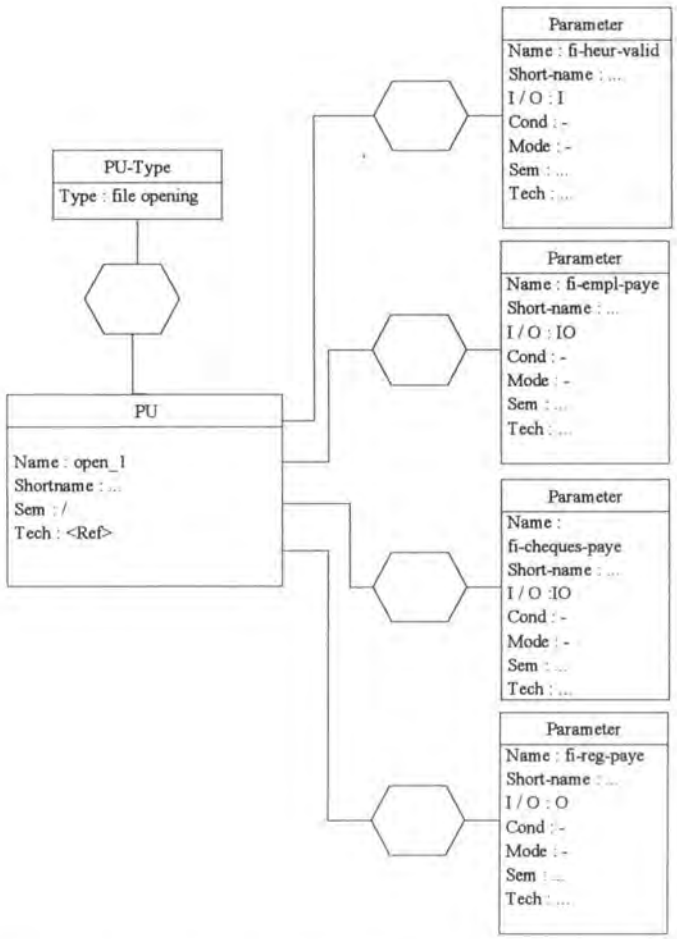


Figure 5-7 : Cobol - Modélisation d'une instruction d'ouverture de fichiers

La seconde instruction que nous allons étudier est l'instruction *accept*.

Cette instruction est représentée par une *processing unit*. Celle-ci est décomposée en deux parties : la première indiquant la variable destinée à contenir l'information (*target*), l'autre spécifiant l'origine de la saisie (*origin*). Dans ce cas-ci, l'origine est la date système (voir Figure 5-8).

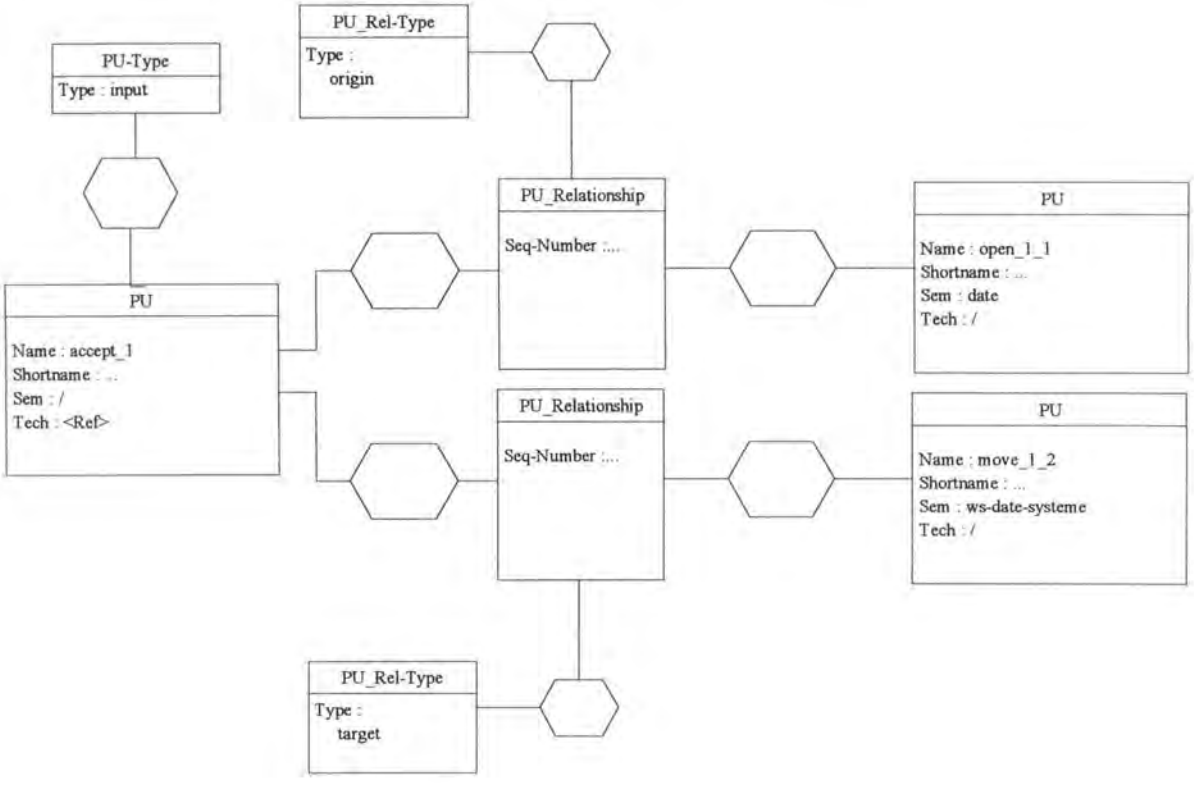


Figure 5-8 : Cobol - Modélisation d'une instruction de saisie

Nous allons maintenant sélectionner quelques instructions particulières.

La première consiste en un branchement conditionnel (voir Exemple 5-7).

```
IF VAL-HEURES GREATER THAN WS-POINT-HSUPP
        COMPUTE WS-BRUT     =       (WS-POINT-HSUPP * MAIT-TAUX)
                                    + ((VAL-HEURES  -  WS-POINT-HSUPP)
                                            * MAIT-TAUX * WS-FACT-H-SUPP)
ELSE
        COMPUTE WS-BRUT = VAL-HEURES * MAIT-TAUX
```

Exemple 5-7 : Cobol - Instruction IF ... ELSE ...

L'instruction constitue une *processing unit*. Celle-ci est décomposée en trois parties, représentant respectivement la condition, les instructions à effectuer si la condition est vérifiée et celles à effectuer dans le cas contraire. Les variables intervenant dans la condition constituent des paramètres de l'entité contenant cette condition (voir Figure 5-9).

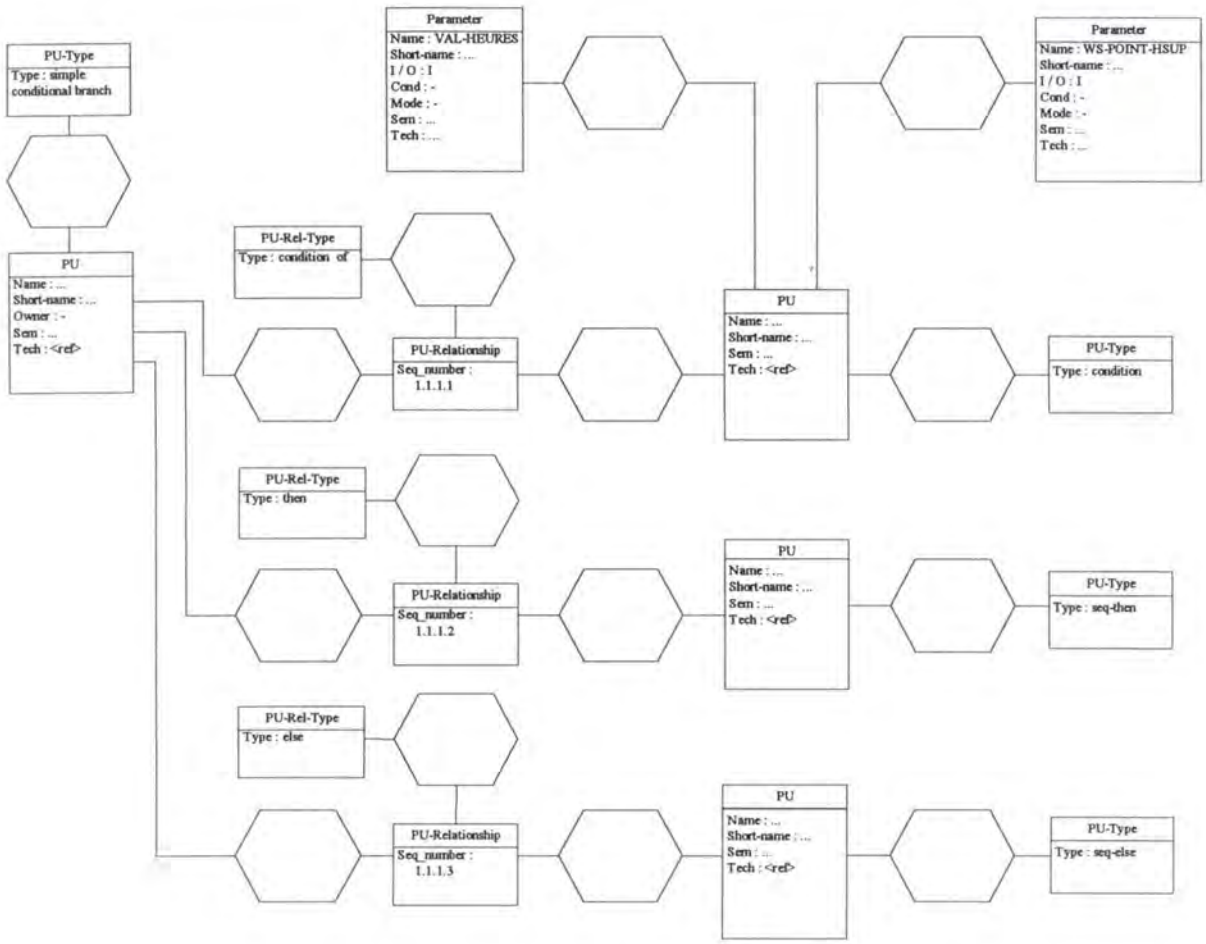


Figure 5-9 : Cobol - Modélisation d'une instruction IF ... ELSE ...

L'instruction se trouvant dans la partie *then* du branchement conditionnel est une instruction d'affectation. Cela se représente en décomposant la *processing unit* en deux sous-*processing units* modélisant l'origine et la destination de l'affectation (voir Figure 5-10).

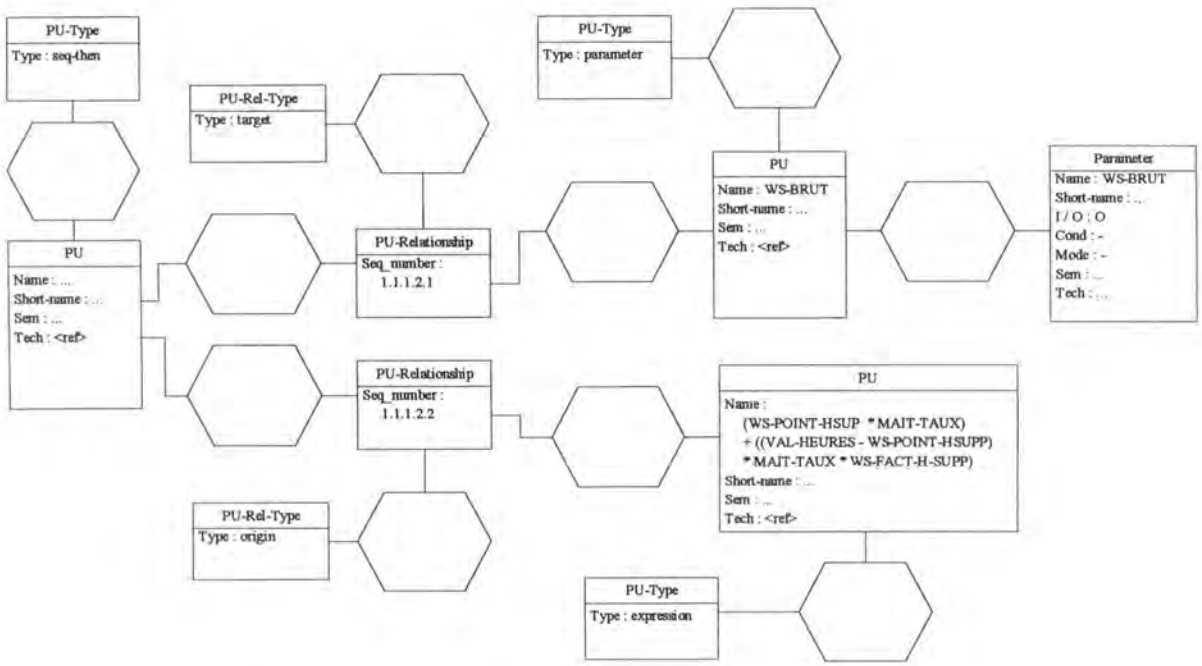


Figure 5-10 : Cobol - Modélisation d'une instruction d'affectation

Nous devons encore décomposer l'expression origine de l'affectation. Pour cela, nous créons à nouveau deux *processing units* représentant les deux sous-expressions concernées par l'opérateur principal de l'expression. Le type de l'opération effectuée sur ces deux sous-expressions est indiqué dans l'entité *PU-Rel-Type* (voir Figure 5-11).

Les *processing units,* référençant les deux sous-expressions, doivent encore être décomposées, mais nous ne l'avons pas représenté sur notre graphique, le procédé à utiliser étant identique.

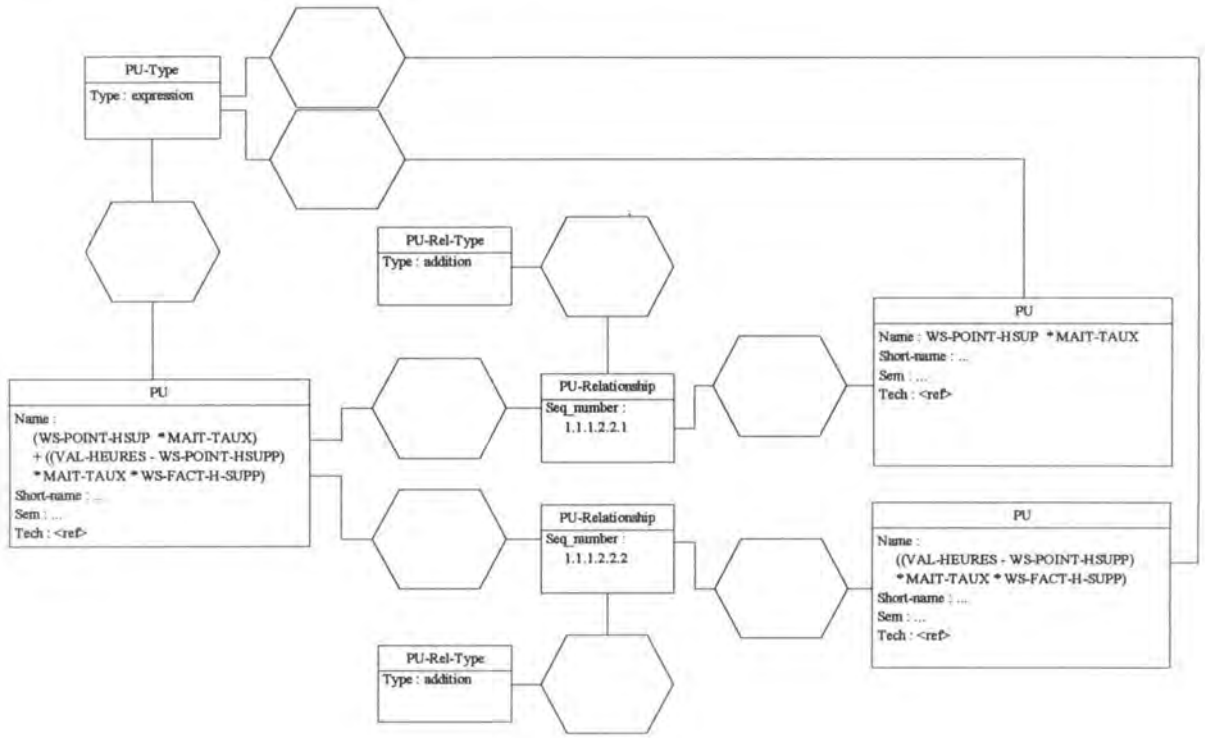


Figure 5-11 : Cobol - Modélisation d'une instruction d'affectation

Nous allons maintenant décomposer une instruction de recherche dans une table (voir Exemple 5-8).

```
SET IND-TAB TO 1
SEARCH TABLE-IMPOTS
        WHEN WS-BRUT GREATER THAN LIM-BASSE
                COMPUTE WS-IMPOT = WS-BRUT * TAUX-IMP(IND-TAB)
```

Exemple 5-8 : Cobol - Instruction de recherche dans une table

Cette instruction peut être considérée comme ayant deux parties. Une partie sert à initialiser l'index qui va être utilisé pendant la recherche, tandis que l'autre effectue la recherche et spécifie l'action à effectuer lorsque le critère de recherche est satisfait (voir Figure 5-12).

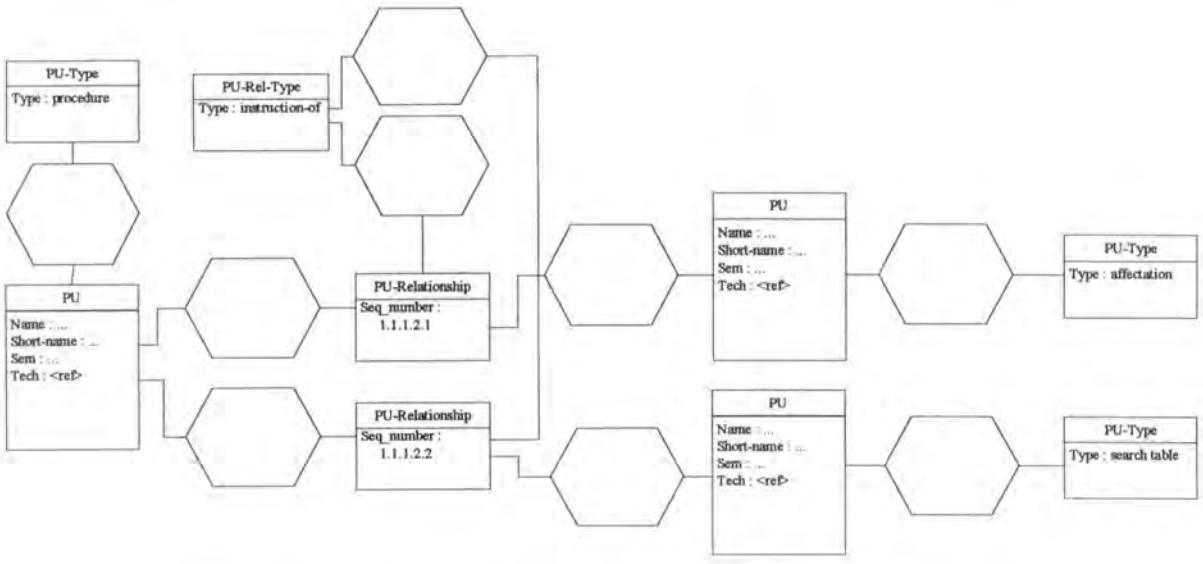


Figure 5-12 : Cobol - Modélisation d'une instruction de recherche dans une table

La seconde *processing unit* doit encore être raffinée. Nous pouvons en effet la redécomposer en deux sous-*processing units* : l'une contenant la condition de recherche, et l'autre indiquant l'action à effectuer lorsque cette recherche est fructueuse (voir Figure 5-13).

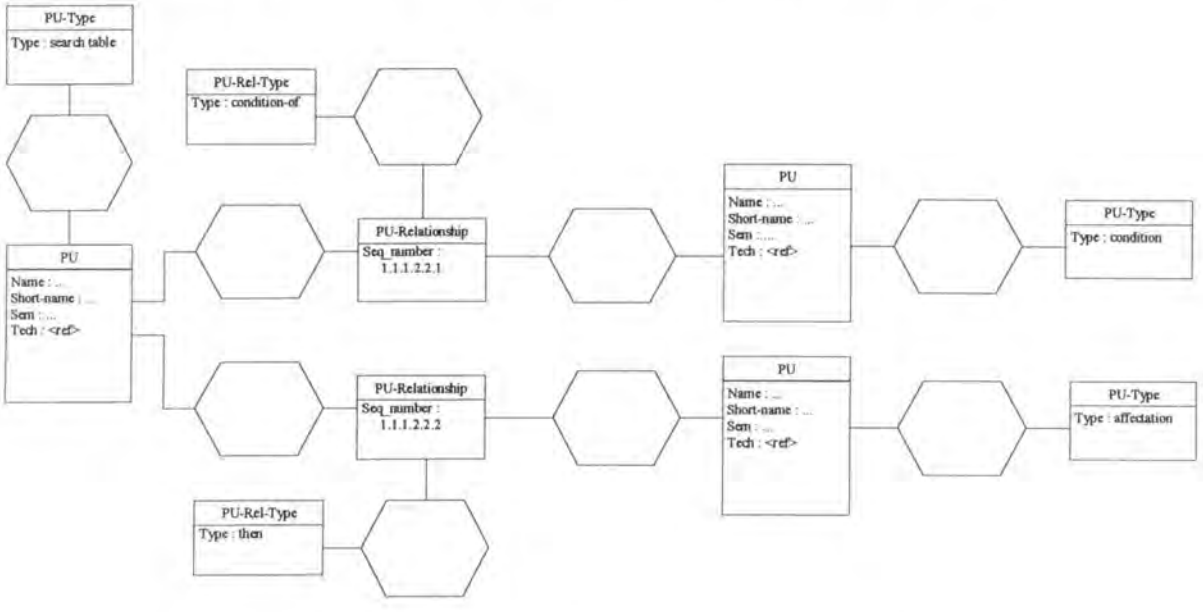


Figure 5-13 : Cobol - Modélisation d'une instruction de recherche dans une table

La dernière instruction que nous allons représenter est une instruction de lecture dans un fichier (voir Exemple 5-9). Cette instruction nous permettra de décomposer également une instruction de type affectation.

```
READ FI-HEUR-VALID
        AT END
                MOVE "OUI" TO WS-FIN-HEUR
                MOVE HIGH-VALUES TO VAL-ID
```

Exemple 5-9 : Cobol - Instruction de lecture dans un fichier

La première opération que nous réalisons consiste à relier la *processing unit* référençant l'instruction avec son paramètre (fichier). Nous relions ensuite cette *processing unit* à deux autres contenant respectivement la condition (*at end*) et les instructions devant être exécutées lorsque celle-ci est remplie (voir Figure 5-14).

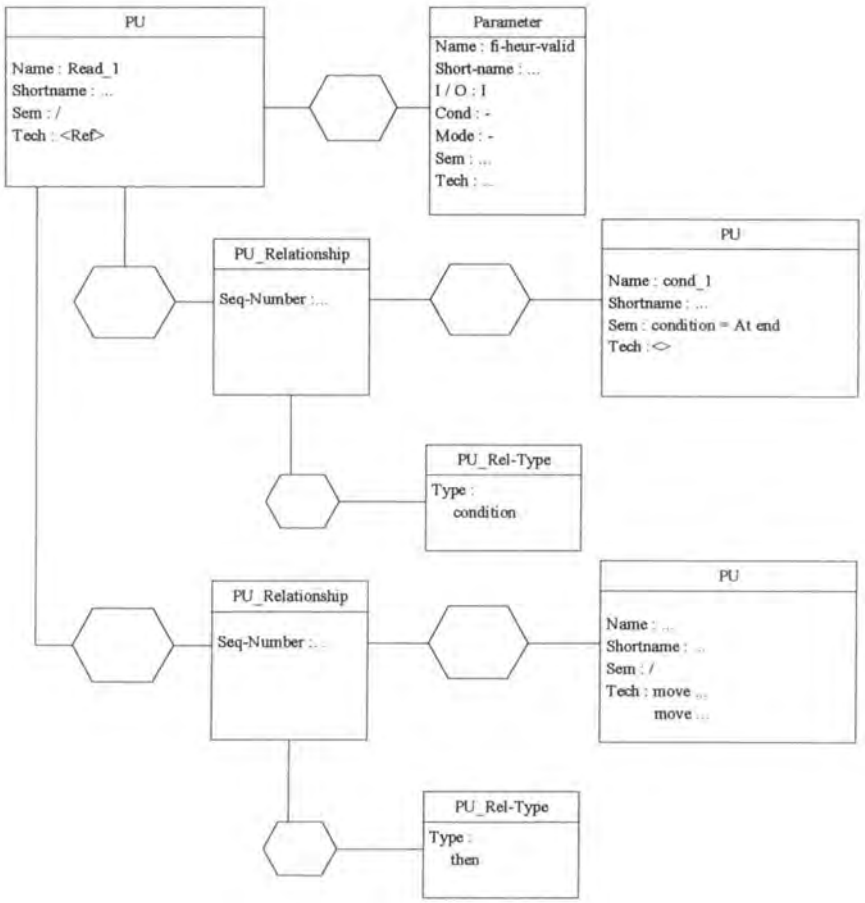


Figure 5-14 : Cobol - Modélisation d'une instruction de lecture dans un fichier

L'entité, contenant une référence vers les instructions à effectuer lorsque la condition est remplie, doit être décomposée en deux *processing units* référençant les deux instructions d'affectation.

Nous allons analyser la représentation de l'une d'entre elles.

La *processing unit* est décomposée en deux sous-*processing units*. La première contient l'origine de l'affectation, et l'autre contient sa destination. Cette destination étant une variable,

nous la relions à une entité *parameter* qui sera elle-même reliée à son *entity-type* (voir Figure 5-15).

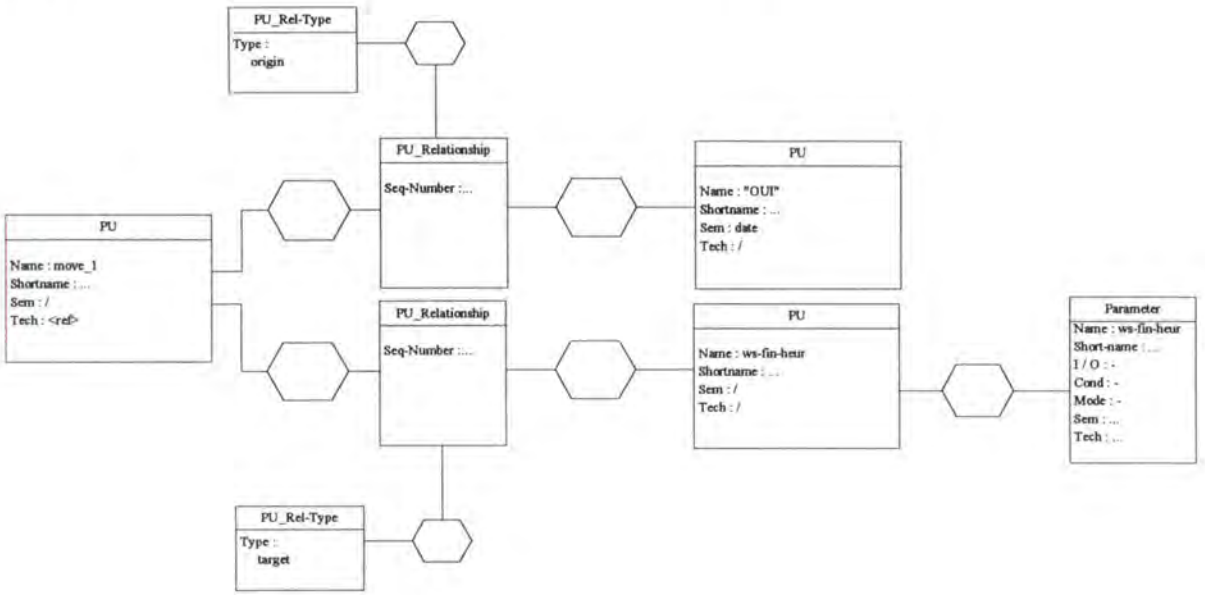


Figure 5-15 : Cobol - Modélisation d'une instruction move

6. Proposition d'une interface

6.1 Introduction

La première section de ce chapitre est dédiée à la présentation des boîtes de dialogue relatives à la saisie des informations devant figurer dans le méta-schéma. Etant donné que les boîtes de dialogue concernant la partie données ont déjà été implémentées au sein de l'atelier DB-Main, nous nous devons de rester cohérents avec celles-ci.

Nous allons donc commencer, dans une première section, par présenter les boîtes de dialogue existantes, avant de poursuivre par la proposition d'une nouvelle série de boîtes référençant la partie traitements du méta-schéma.

La dernière section de ce chapitre sera consacrée à la proposition de représentations des données, à savoir une représentation sous forme textuelle et une sous forme graphique.

6.2 *Boîtes de dialogue*

6.2.1 Saisie de la partie données

Avant toute opération, l'utilisateur doit créer un nouveau schéma de données. Pour cela, il a à sa disposition la boîte de dialogue suivante (voir Figure 6-1) :

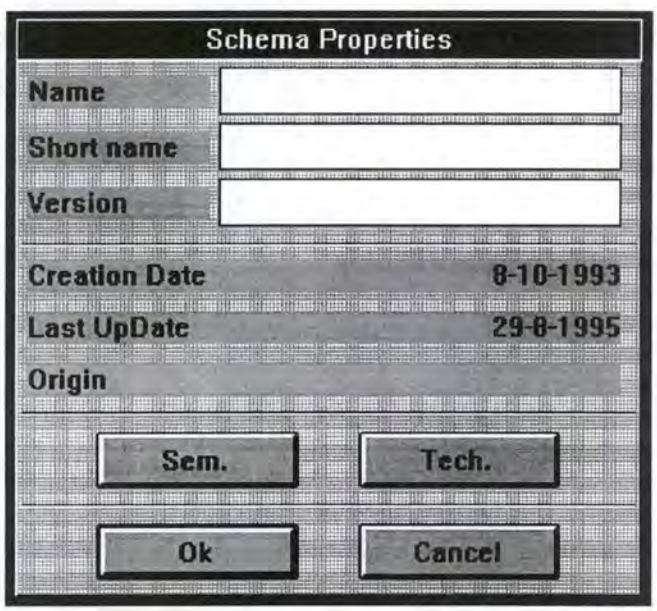


Figure 6-1 : Création d'un schéma de données

Il doit donc indiquer le nom, le nom court et le numéro de version de son schéma. Des informations sémantiques ou techniques relatives au schéma peuvent être stockées grâce aux boutons *Sem* et *Tech.*

Une fois le schéma des données créé, l'utilisateur peut ajouter des entités, créer des relations entre ces entités, ...

Pour créer un nouveau type d'entité, la boîte de dialogue suivante sera affichée (voir Figure 6-2).

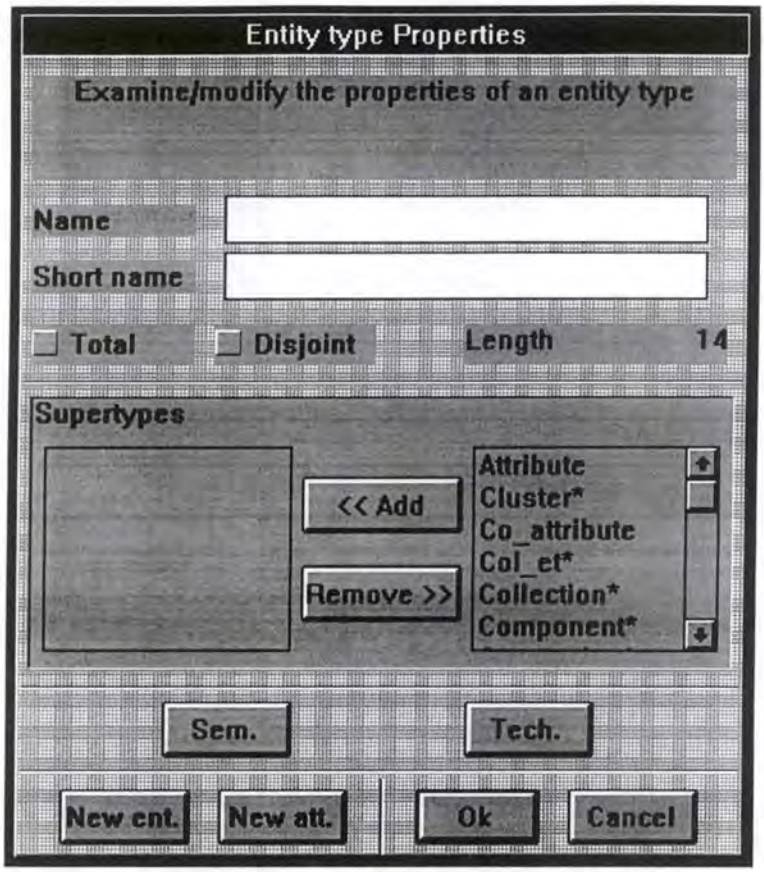


Figure 6-2 : Création d'une Entity-type

Outre le nom et le nom court du type d'entité, l'utilisateur peut, en plus, indiquer le fait qu'un type d'entité est un sur-type d'un autre. Pour cela, la liste des types d'entité déjà créés est affichée dans une liste défilante. Il suffit donc de faire son choix parmi ceux-ci. A nouveau, des informations sémantiques et techniques peuvent être ajoutées.

La création d'une relation se fait, elle-aussi, au moyen d'une boîte de dialogue (voir Figure 6-3). Le nom et le nom court de la relation sont saisis. L'utilisateur a alors la possiblité de créer les rôles que cette relation va jouer.

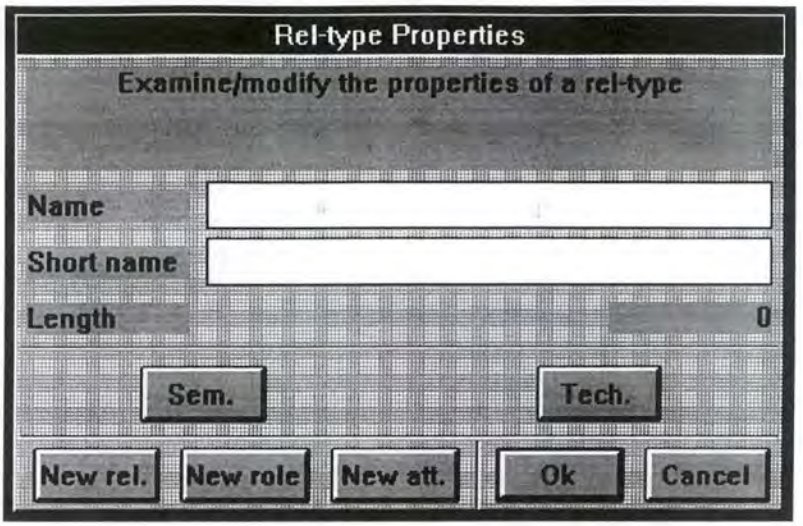


Figure 6-3 : Création d'une relation

Un rôle possède un nom et une cardinalité. Il doit bien sûr être relié à un type d'entité. A nouveau, une liste défilante contenant l'ensemble des types d'entité déjà créés est affichée, et il suffit de sélectionner l'entité désirée (voir Figure 6-4).

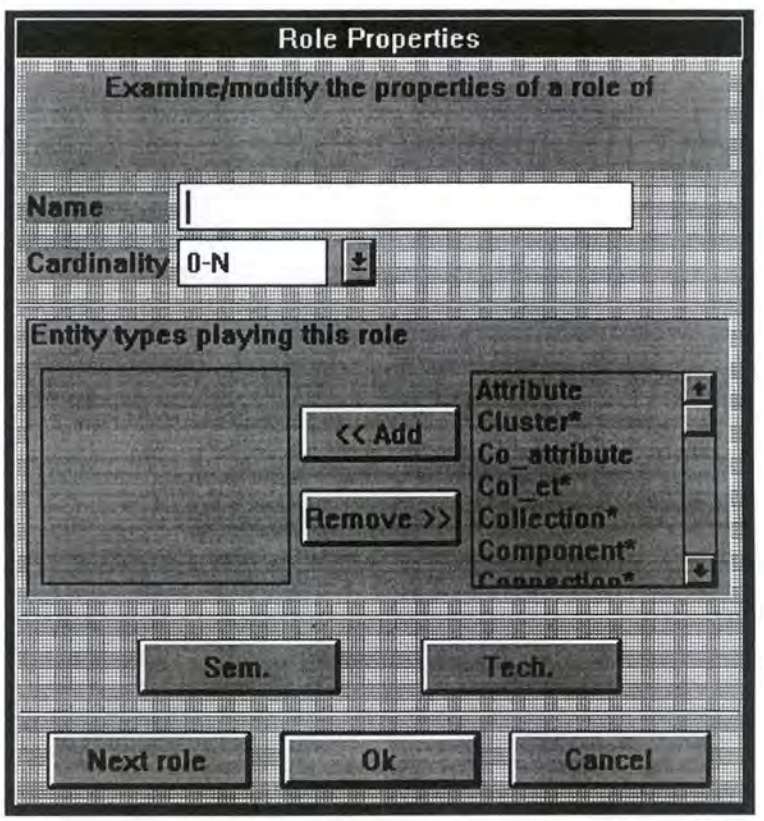


Figure 6-4 : Création d'un rôle

Un type d'entité ou une relation peut possèder un certain nombre d'attributs. Pour ajouter un attribut à l'un de ces deux objets, il suffit de cliquer sur le bouton *New att.* afin de voir apparaître la boîte de dialogue suivante (voir Figure 6-5) :

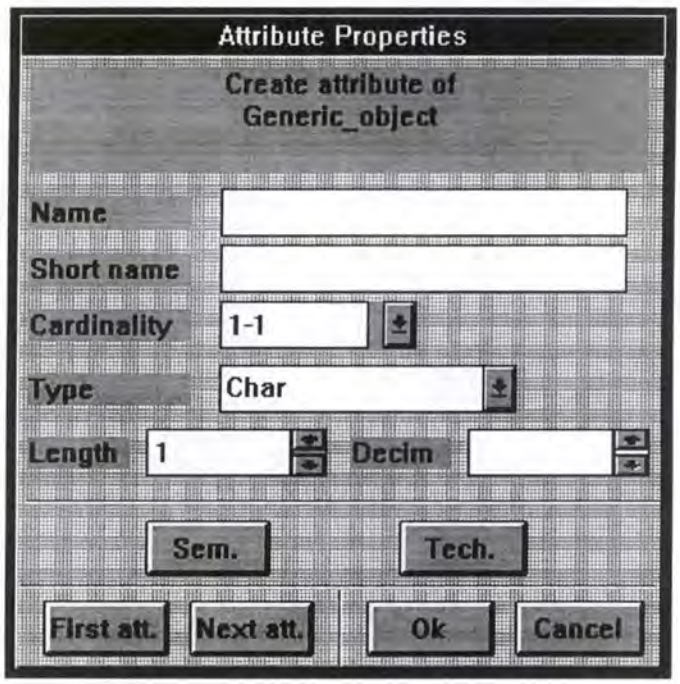


Figure 6-5 : Création d'un attribut

Après avoir saisi le nom et le nom court de l'attribut, nous pouvons indiquer sa cardinalité. Cela est nécessaire pour représenter un attribut multivalué. Une liste de cardinalités est fournie à l'utilisateur, mais rien ne l'empêche d'indiquer une cardinalité non présente dans cette liste. Par contre, l'utilisateur est tenu de choisir un type présent dans la liste.

Le dernier écran que nous allons présenter est relatif à la notion de groupe. C'est cette notion qui permet d'indiquer des concepts tels que l'identifiant primaire ou secondaire, la contrainte d'exclusion, ... (voir Figure 6-6).
L'utilisateur indique tout d'abord le nom du groupe. Il choisit ensuite, au sein d'une liste défilante, les attributs et / ou les rôles qui sont concernés par la contrainte avant de spécifier le type de contrainte envisagé.

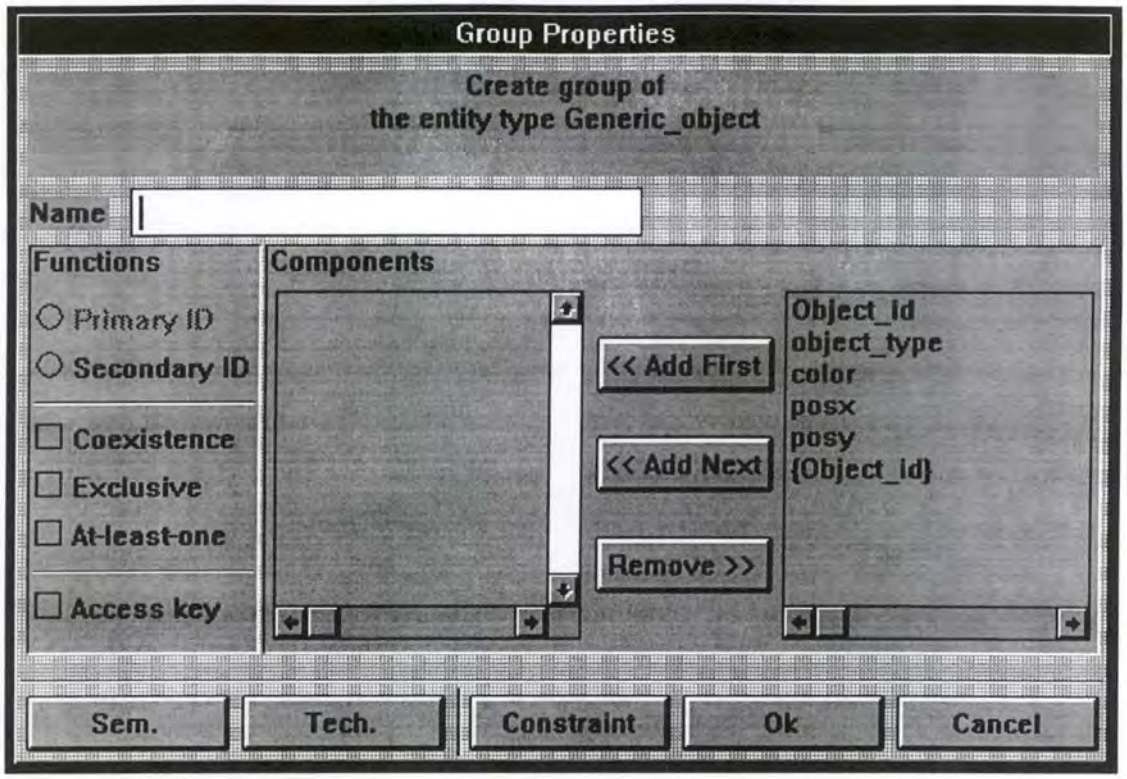


Figure 6-6 : Création d'un group

6.2.2 Saisie de la partie traitements

La première action que l'utilisateur devra effectuer avant de créer ses traitements est la création d'un *PU Schéma*. Cela se fait au moyen de la boîte de dialogue suivante (voir Figure 6-7).

La seule différence par rapport à la création d'un schéma de données est le champ *Model*, qui a pour but de saisir le modèle dans lequel sont exprimés les traitements.

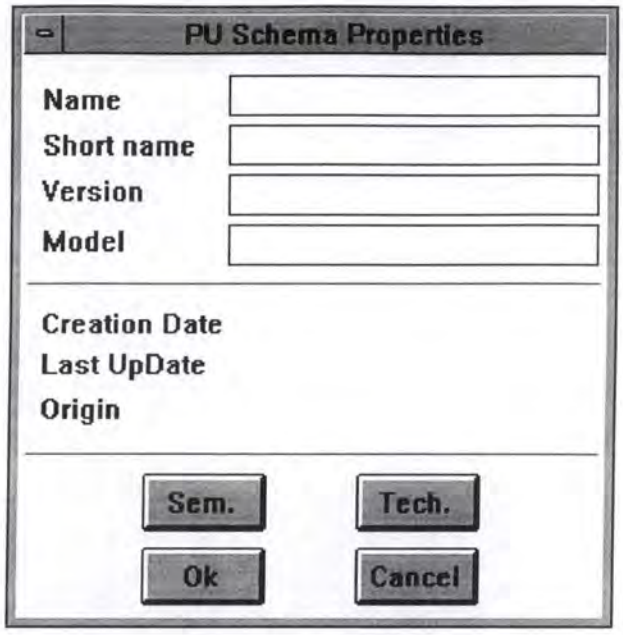


Figure 6-7 : Création d'un PU-Schema

La boite de dialogue suivante (voir Figure 6-8) permet à l'utilisateur de créer une nouvelle *Processing Unit*. La liste *Type* contiendra tous les types que peut possèder une *Processing Unit*. Il suffit de faire une choix parmi ces types ; ce choix est obligatoire.
La liste défilante *Actor* contient l'ensemble des acteurs qui ont déjà été créés. Dès qu'un acteur a été sélectionné dans cette liste, la liste *Role* devient active. Elle permet de définir le rôle que l'utilisateur joue par rapport à cette *Processing Unit*.

Le bouton *New PU* permet de passer directement à la créaction d'une nouvelle *Processing Unit*, tandis que le bouton *New par.* permet d'assigner certains paramètres à cette *Processing Unit*.

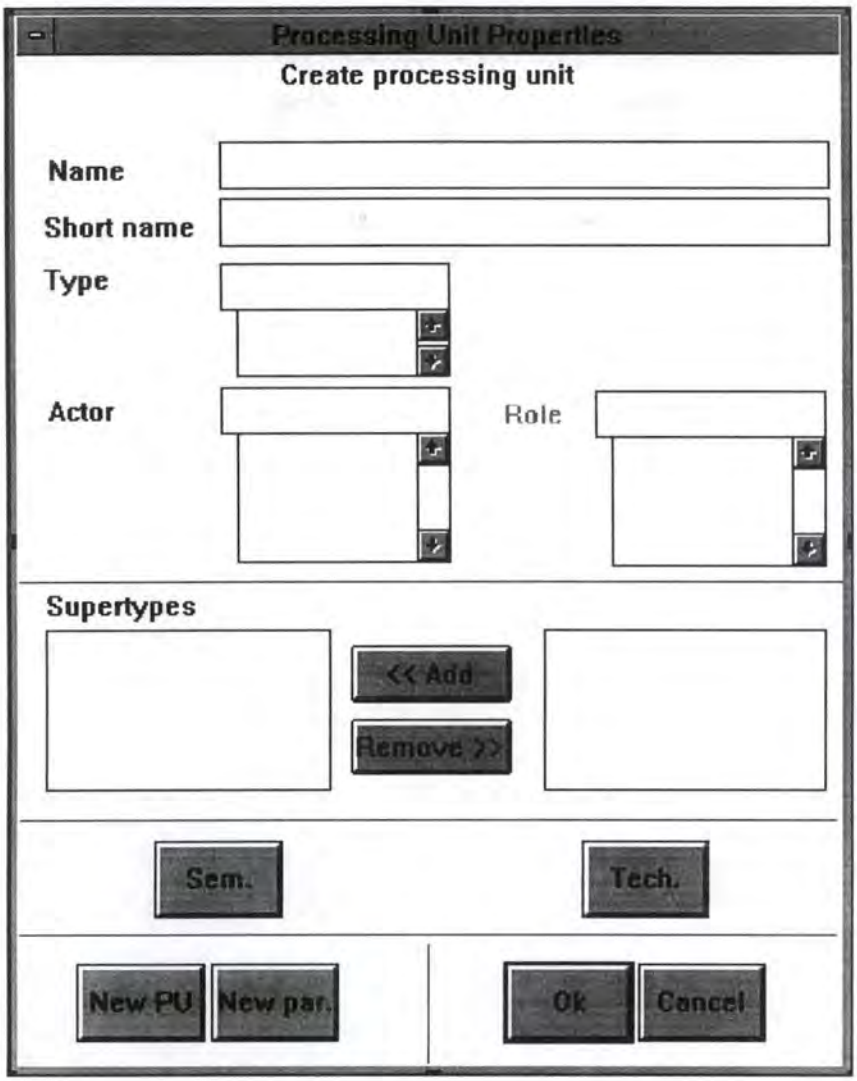


Figure 6-8 : Création d'une processing unit

Cette boîte de dialogue (voir Figure 6-9) permet à l'utilisateur d'attacher certains paramètres à une *Processing unit*. Pour cela, l'utilisateur doit choisir dans la liste qui lui est proposée le nom d'une *Entity type* qui lui sera proposé. Il devra ensuite indiquer si ce paramètre est un paramètre en entrée ou en sortie de la *Processing Unit*, ainsi que le mode de passage de ce paramètre. La sélection d'un acteur ainsi que son rôle se fait comme dans le cas de la création d'une *Processing Unit*.

Le champ de saisie *condition* sert, quant à lui, à indiquer la condition qui doit être vérifiée pour que ce paramètre soit passé à la *Processing Unit*.

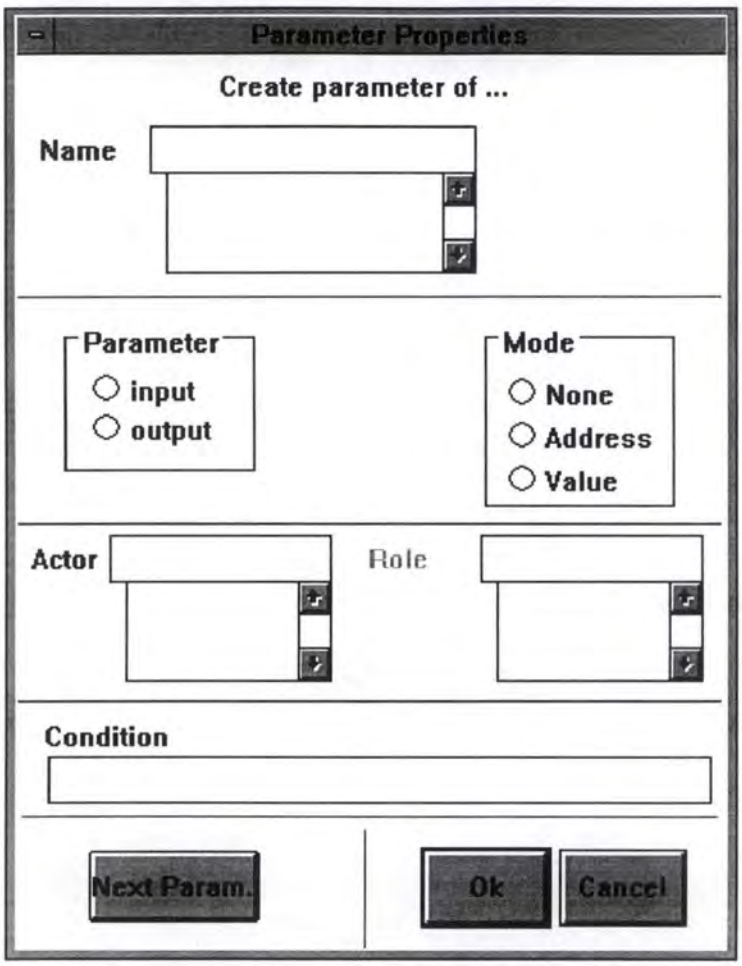


Figure 6-9 : Création de paramètres

L'utilisateur a la possibilité de créer de nouveaux acteurs (voir Figure 6-10). Cela sera nécessaire si l'on veut lui attribuer un paramètre ou une processing unit.

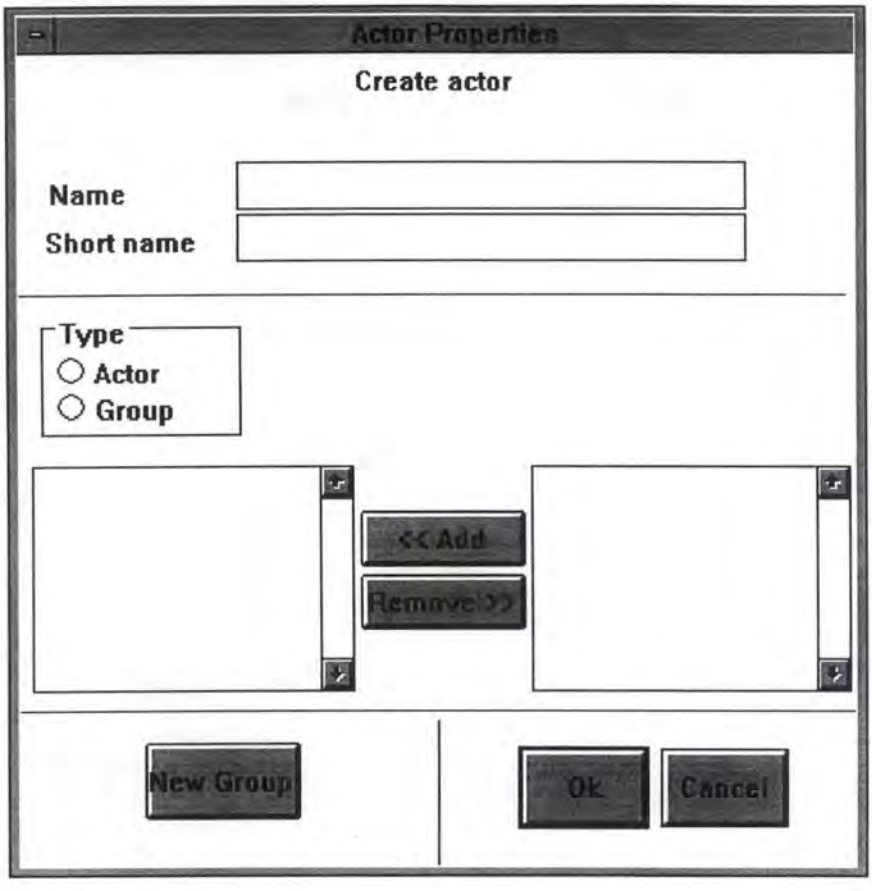


Figure 6-10 : Création d'acteurs

Nous pouvons également créer un groupe d'acteurs. La liste des acteurs déjà créés est présente dans la liste défilante de droite. Il suffit alors à l'utilisateur de choisir ceux qui vont appartenir à son groupe.

L'utilisateur a bien sûr la possibilité de créer des relations entre les différentes *Processing Units* qu'il a créées (voir Figure 6-11). Pour cela, il choisit la *processing unit* source. A ce moment, la liste des *processing units* qui sont déjà en relation avec elle apparaît dans la liste prévue à cet effet ; cette liste est triée par ordre de type.

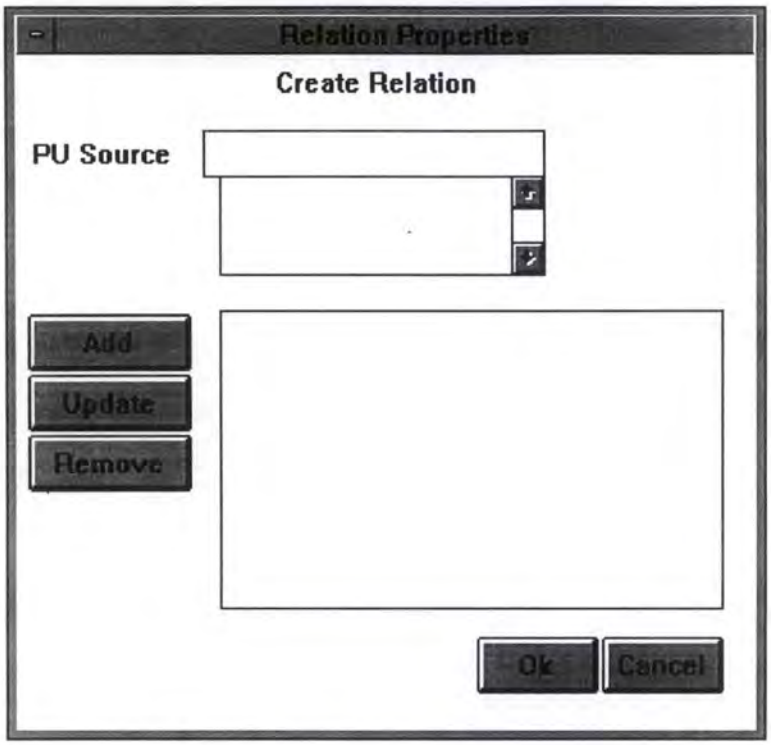


Figure 6-11 : Création d'une relation entre processing units

Une fois cette opération réalisée, l'utilisateur peut soit ajouter une nouvelle relation, soit modifier ou supprimer une relation existante.

La sélection d'une de ces actions entraîne l'affichage d'une seconde boîte de dialogue (voir Figure 6-12).

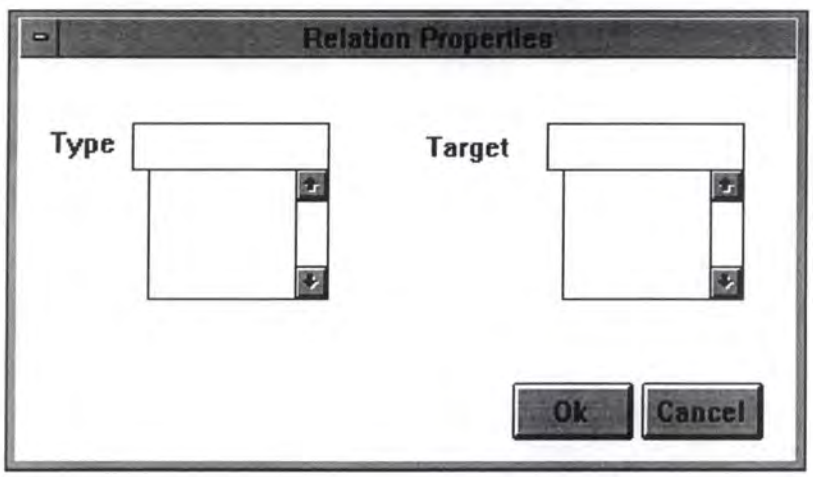


Figure 6-12 : Création d'une relation entre processing units

6.3 *Représentations des spécifications*

Dans cette partie, nous allons proposer une représentation textuelle ainsi qu'une représentation graphique de schémas de traitement.

6.3.1 Vue textuelle

Il s'agit ici d'afficher à l'écran, sous forme textuelle, toutes les informations (ou seulement une partie d'entre elles) relatives aux différents objets présents dans le repository. Pour chaque objet, nous partirons d'un exemple complet couvrant tous les cas possibles (voir Figure 6-13) et nous montrerons la façon dont nous proposons de le représenter.

Nous représenterons 4 objets : le *PU_Schema*, les *processing units*, les *actors* et les *parameters*.

Exemple

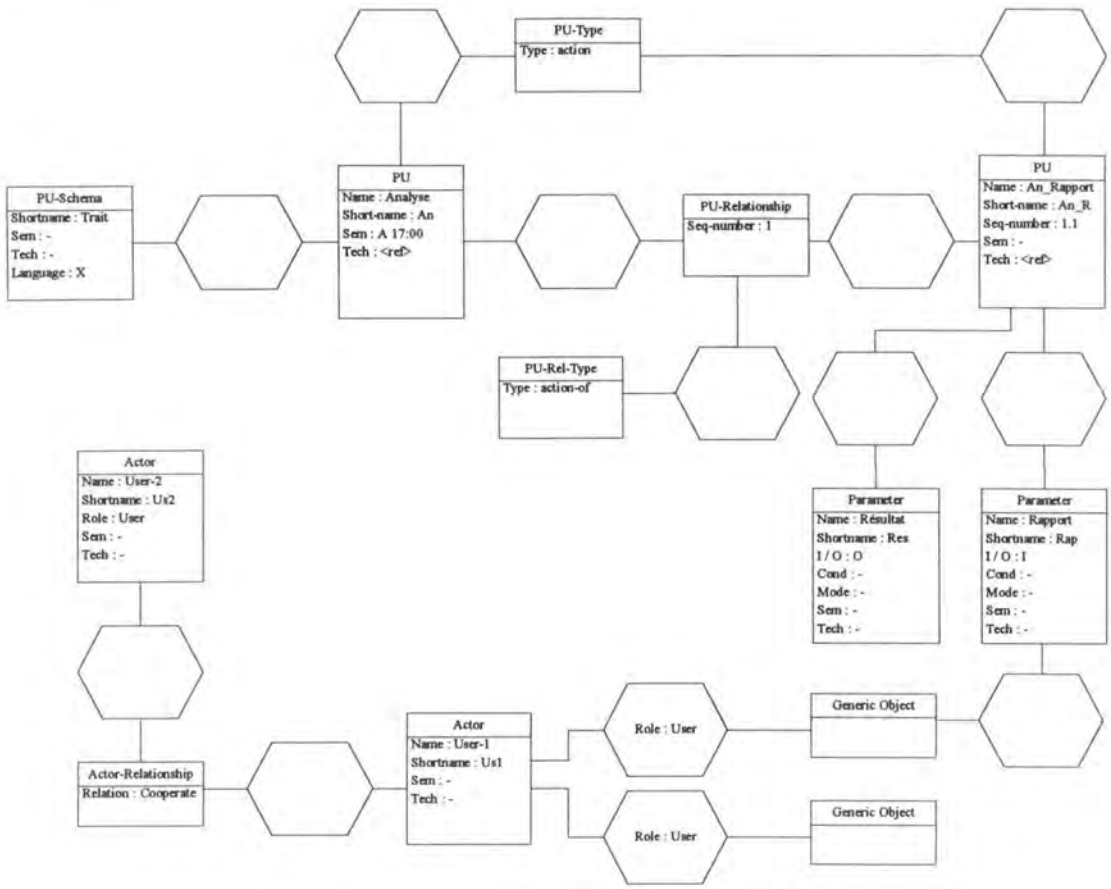


Figure 6-13 : Exemple global

6.3.1.1 PU_Schema.

Un *PU_Schema* a pour représentation une ligne reprenant son nom ainsi que le modèle (voir Figure 6-14).

PU Schema : Trait - X

Figure 6-14 : Représentation textuelle d'un PU_Schema

6.3.1.2 Processing unit.

Pour représenter une *processing unit*, nous indiquerons (sur la même ligne) son nom, son nom court, son type ainsi qu'une remarque précisant s'il existe une description sémantique ([S]) et /

ou une description technique ([T]). Ensuite les relations avec d'autres *processing units*, les *parameters* ainsi que les *actors* seront indiqués (voir Figure 6-15).

PU Schema : Trait - X

Analyse / An [S / T] : Action
relation : Action-of An_Rapport

An_rapport / An_R [T] : Action
relation : Action-of Analyse
parameter : Input Rapport
Output Résultat
actor : User-1

Figure 6-15 : Représentation textuelle de processing units

6.3.1.3 Parameters.

Notre principe reste le même pour les *parameters*. Nous allons indiquer la valeur de ses attributs et, ensuite, une référence aux *generic-objects*, *processing units* et aux *actors* qui interviennent dans le *parameter* considéré (voir Figure 6-16).

PU Schema : Trait - X

Analyse / An [S / T] : Action
relation : Action-of An_Rapport

An_rapport / An_R [T] : Action
relation : Action-of Analyse
parameter : Input Rapport
Output Résultat
actor : User-1

Rapport / Rap : In

generic object : Rapport

processing unit : An_Rapport

actor : User-1

Résultat / Res : Out

generic object : Résultat

processing unit : An_Rapport

Figure 6-16 : Représentation textuelle de parameters

6.3.1.4 Actor.

Pour ce dernier objet, nous allons indiquer la valeur de ses attributs, une référence aux *processing units* et *parameters* dans lesquels il joue un rôle ainsi qu'une référence aux *actors* qui ont une relation avec l'*actor* considéré (voir Figure 6-17).

PU Schema : Trait - X

Analyse / An [S / T] : Action

relation : Action-of An_Rapport

An_rapport / An_R [T] : Action

relation : Action-of Analyse

parameter : Input Rapport

Output Résultat

actor : User-1

Rapport / Rap : In

generic object : Rapport

processing unit : An_Rapport

actor : User-1

Résultat / Res : Out
generic object : Résultat
processing unit : An_Rapport

User-1 / Us1 : User
actor : User-2
processing unit : An_Rapport
parameter : Rapport

User-2 / Us2 : User
actor : User-1

Figure 6-17 : Représentation textuelle d'acteurs

6.3.2 Vue graphique

La représentation graphique ne peut pas être, comme la représentation textuelle, indépendante du modèle. En effet, bon nombre de modèles possèdent leur propre représentation et nous nous efforcerons de la conserver. Dans ce cas, selon le modèle, une représentation particulière sera utilisée. D'autres modèles, par contre, ne possèdent pas une représentation propre. Nous en proposerons donc une en tentant de ne pas trop nous éloigner des représentations prédéfinies.

Tout comme dans la représentation graphique du schéma des données, nous représenterons le schéma par un ovale dans lequel sera inscrit le nom du *PU-Schema*. De plus, dans ce même ovale, nous afficherons le modèle dans lequel est exprimé le schéma.

Nous proposons ensuite de représenter les *processing units* par des rectangles. Deux *processing units* ayant entre elles une relation seront reliées par un arc dont le nom sera le type de relation existant entre ces deux *processing units*.

Un *parameter* sera représenté par ce symbole :

Le sens de la flèche reliant la *processing unit* à son *parameter* indiquera si le *parameter* est une entrée, une sortie ou les deux à la fois.

Un acteur effectuant une *processing unit* ou possédant une donnée verra son nom affiché à l'écran dans un losange. Ce losange sera relié à l'objet correspondant par un arc non orienté.

Maintenant que nous avons défini les symboles graphiques correspondant aux différents objets, reprenons l'exemple précédent (voir Figure 6-13) et voyons ce que donne notre représentation (voir Figure 6-18).

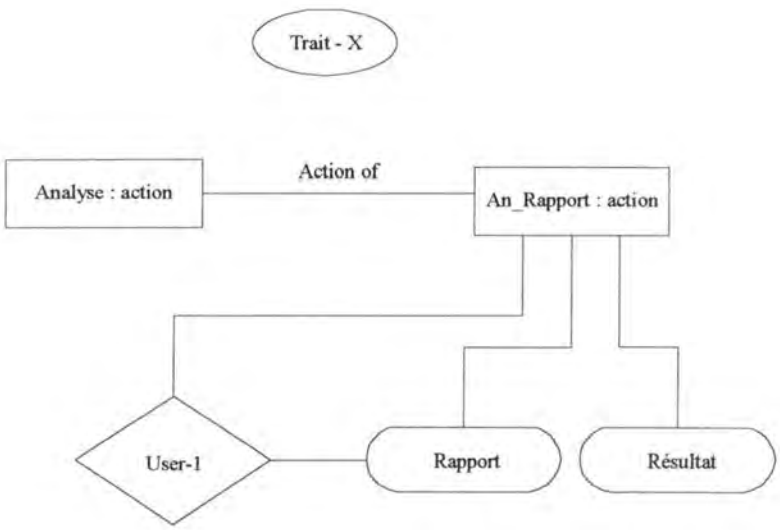


Figure 6-18 : Représentation graphique

7. Conclusion

Au cours de ce mémoire, nous avons élaboré et validé un méta-schéma permettant la représentation d'applications de tout type. Cela inclut une modélisation de l'ensemble des données et des traitements présents dans ces applications.

Un méta-schéma existe déjà au sein de l'outil CASE DB-Main mais il ne s'intéresse qu'à la partie données des applications. Nous avons donc proposé une extension relative à la partie traitements. Certaines informations ne pouvant être représentées dans le méta-schéma actuel nous avons été contraints d'y apporter quelques modifications. Toutefois, nous avons tenté de réduire au maximum l'ampleur de ces modifications.

Certaines des solutions que nous avons proposées peuvent sembler résulter d'une nécessité propre à un modèle particulier plutôt que d'une analyse combinée des différents modèles. Néanmoins nous avons dû nous y résoudre afin, comme nous l'avons précisé dans le paragraphe précédent, de minimiser les changements à apporter aux structures actuelles et de

produire un méta-schéma aussi générique que possible. En effet, certains modèles ont des structures qui leur sont propres et qui peuvent difficilement s'insérer dans un modèle classique.

Enfin, il nous semble utile, en cas d'améliorations ultérieures, de pousser un peu plus loin l'analyse de la représentation textuelle d'une application afin de permettre à l'utilisateur d'indiquer le niveau de détail des informations qui lui seront fournies.

8. Bibliographie

[ACH93] W., Achtert,
Le Grand Livre du C++,
Micro Application, 1993.

[BYR??] Byron / S. Gottfried
Programmation C.

[COA??] P., Coad / E., Yourdon,
Object-Oriented Analysis
Yourdon Press, Prentice Hall.

[COL86] Collongues, Alain / Hugues, Jean / Laroche, Bernard
Merise : méthode de conception
Dunod, Paris, 1986.

[DBM94] V., Englebert / J., Henrard / J-M., Hick / D., Roland
Description du méta-schéma dans une perspective orienté objet de l'atelier logiciel "DB-MAIN"
Projet DB-Main, Spécification - SPEC-94/3-1.

[ELM94] R., Elmasri / S. B., Navathe,
Fundamentals of Database Systems
The Benjamin/Cummings Publishing Company Inc., 1994.

[FIE92] B., Fields,
A guide to reading VDM specifications
University of Manchester, Technical Report Series UMCS-92-12-4.

[GES89] N., Gestalder / J., Jaray
VDM
in A First Collection of Case Studies, ICARUS, Juin 1989.

[HAY93] J.J., Hayes / C.B., Jones / J.E., Nicholls,
Understanding the differences between VDM and Z
University of Manchester, Technical Report UMCS-93-8-1.

[ICL??] ICL
ProcessWise Integrator
PML Reference Manual

[JAR??] M., Jarke
Database Application Engineering with DAIDA
Research Reports ESPRIT, DAIDA Vol. 1.

[MDL94] E., Dubois,
Cours de Méthodologie de développement de logiciels,
FUNDP Namur, 1994.

[MYL90] J., Mylopoulos, A., Borgida, M., Jarke, M., Koubarakis,
Telos : Representing Knowledge About information Systems,
ACM Transactions on Information Systems, Vol. 8, N° 4, October 1990, Pages 325-362.

[MYL95] J., Mylopoulos,
Chaire Internationale, Conceptual Modeling for Information Systems Engineering,
Leture Series presented at the University of Namur, April 24 - May 4, 1995

[NEW86] Lawrence R., Newcomer,
Programmation en Cobol Structuré, Théorie et Applications,
Traduit de : Schaum's outline series, Theory and problems of PROGRAMMING WITH STRUCTURED COBOL by Lawrence R. Newcomer,
McGraw-Hill, Paris, 1986.

[PML94] M., Spink
PML Booklet, Incomplete Draft Version
PEVE IT Unit, Department of Computer Science, University of Manchester, July 1994.

[SER94] A., Sernadas / J. F., Costa / C., Sernadas,
Object Specification Through Diagrams, OBLOG Approach,
Computer Science Section, Mathematics Department, Instituto Superior Técnico & Computer Science Group INESC, 1994.

[ZEI95] J-M., Zeippen,
OBLOG - An Object-Oriented, formal, tool-based software engineering approach
Namur, March-April 1995.